Lighting and Shading Techniques for Interactive Applications

Organizer:
David Blythe
Silicon Graphics

August 8, 1999

SIGGRAPH '99 Course 12

Abstract

This advanced course demonstrates sophisticated and novel techniques for lighting and shading scenes in interactive applications using the widely available OpenGL graphics library.

The course focuses on traditional lighting and shading techniques and discusses how they are implemented on current graphics hardware. The course then explores how these techniques can be effectively extended to increase realism while maintaining interactive performance. The application areas include: entertainment and visual simulation, CAD, and scientific visualization.

We use OpenGL as our rendering platform, but the theory and algorithms described in the course can be applied to other rendering architectures and APIs.

Advanced Graphics Programming Techniques Using OpenGL

Organizer:
David Blythe
Silicon Graphics

August 9, 1999

SIGGRAPH '99 Course 29

Abstract

This advanced course demonstrates sophisticated and novel computer graphics programming techniques, implemented in C using the widely available OpenGL library.

By explaining the concepts and demonstrating the techniques required to generate images of greater realism and utility, the course helps students achieve two goals: they gain a deeper insight into OpenGL functionality and computer graphics concepts, while expanding their "toolbox" of useful OpenGL techniques.

Speakers

David Blythe

David Blythe is a Principal Engineer in the Advanced Graphics Software group at Silicon Graphics. David joined SGI in 1991 and has contributed to the development of the RealityEngine and InfiniteReality graphics systems. He has worked extensively on implementations of the OpenGL graphics library and OpenGL extension specifications. David is currently working on high-level toolkits which are built on top of OpenGL as well as contributing to the continuing evolution of OpenGL. His other interests include large-scale system design and interactive photorealism. David has been a course presenter at SIGGRAPH '96, '97, and '98 as well as other technical forums.

Prior to joining SGI, David was a visualization scientist at the Ontario Centre for Large Scale Computation and a lecturer at the University of Toronto. David received both a B.S. and M.S. degree in computer science from the University of Toronto.

Email: blythe@sgi.com

Brad Grantham

Brad Grantham contributes to the advancement of Linux, OpenGL, and 3D graphics at VA Research, Inc. Brad is also an Adjunct Lecturer at Santa Clara University, where he specializes in helping students write interactive graphics applications.

Brad was a course presenter at SIGGRAPH '97 and '98, and previously contributed at Silicon Graphics to the design and implementation of high-level graphics toolkits, including the Fahrenheit Scene Graph, OpenGL Optimizer, and IRIS Performer. Brad's prior experience also includes UNIX kernel code and imaging codecs. Brad received a B.S. in Computer Science from Virginia Tech in 1992.

Email: grantham@hadron.org

Mark J. Kilgard

Mark J. Kilgard is a Graphics Software Engineer at NVIDIA Corporation where he works on upcoming RIVA graphics processors. Mark authored the book *Programming OpenGL for the X Window System* and implemented the popular OpenGL Utility Toolkit (GLUT) for developing portable OpenGL examples and demos. Previously, Mark worked at Silicon Graphics on the Onyx InfiniteReality graphics supercomputer and on SGI's X Window System implementation. Mark has taught many courses at SIGGRAPH, the Computer Game Developers Conference, and other conferences. Mark's Karaoke rendition of Dolly Parton's "9 to 5" can't be beat.

Email: mjk@nvidia.com

Tom McReynolds

Tom McReynolds is a software engineer at Gigapixel Inc., a company developing Computer Graphics Intellectual Property. He currently works on computer graphics hardware simulation, verification, and device driver software.

Before that, he worked in the Advanced Graphics Software group at Silicon Graphics. He has implemented OpenGL extensions, done OpenGL performance work, and worked on IRIS Performer, a real-time visualization library that uses OpenGL.

Prior to SGI, he worked at Sun Microsystems, where he helped develop graphics hardware support software and graphics libraries, including XGL.

Tom also works as an adjunct professor at Santa Clara University, where he teaches courses in computer graphics using the OpenGL library. He has also presented at the X Technical Conference, was a course organizer and presenter at SIGGRAPH '96, '97, and '98, and presented for SGI at their 1996 Developer Forum, and at SGI's 1997 OpenGL Developer's Workshop.

Email: tomcat@gigapixel.com

Scott R. Nelson

Scott Nelson is a Principal Engineer at Intel doing research on 3D graphics architectures in the Microcomputer Research Labs. Before moving to Intel, Scott spent more than ten years at Sun Microsystems developing 3D graphics accelerator architectures. He contributed to the development of the GT, ZX, and Elite3D graphics accelerators. Before Sun, Scott worked for eight years at Evans & Sutherland developing graphics hardware. He received his B.S. degree in Computer Science from the University of Utah. Scott was a course organizer and presenter at SIGGRAPH '91 and a course presenter at SIGGRAPH '98.

Email: Scott.R.Nelson@intel.com

Other Contributers

Celeste Fowler (Author)

Celeste Fowler is a software engineer in the Advanced Systems Division at Silicon Graphics. She worked on the OpenGL imaging pipeline for the InfiniteReality graphics system and on the OpenGL display list implementation for InfiniteReality and RealityEngine.

Before coming to SGI, Celeste attended Princeton University where she did research on radiosity techniques and TA'd courses in computer graphics and programming systems.

Email: celeste@sgi.com

Simon Hui (Author)

Simon Hui is a software engineer at 3Dfx Interactive, Inc. He currently works on OpenGL and other graphics libraries for PC and consumer platforms.

Prior to joining 3Dfx, Simon worked on IRIS Performer, a realtime graphics toolkit, in the Advanced Systems Division at Silicon Graphics. He has also worked on OpenGL implementations for the RealityEngine and Infinite-Reality. Simon received a B.A. in Computer Science from the University of California at Berkeley.

Email: simon@3dfx.com

Paula Womack (Author)

Paula Womack is a software engineer in the Advanced Systems Division at Silicon Graphics. She has managed the OpenGL group at Silicon Graphics, and was also a member of the OpenGL Architectural Review Board (the OpenGL ARB) that is responsible for defining and enhancing OpenGL.

Prior to joining Silicon Graphics, Paula worked on OpenGL at Kubota and Digital Equipment. She has a B.S. in Computer Engineering from the University of California at San Diego.

Email: womack@sgi.com

Linda Rae Sande (Production Editor)

Linda Rae Sande is a production editor in Technical Publications at Silicon Graphics. A graduate of Northern Arizona University (B.S. in Physics-Astronomy), she has taught college algrebra and physical science courses and worked in marketing communications and technical training. As co-author of two physics laboratory textbooks and author of several production manuals, Linda Rae has many years of experience in book production and production coordination.

Prior to SGI, she was a production coordinator at ESL-TRW responsible for the TravInfo and TransCal transportation project documentation and deliverables.

Email: lindarae@sgi.com

Dany Galgani (Illustrator)

Dany Galgani has provided illustrations to Technical Publications at Silicon Graphics for over 9 years. He has illustrated hardware and software manuals, from user's guides to programmer's manuals.

Before that, he did commercial art for advertising agencies and book publishers, including illustrating books in Ortho's “Do-It-Yourself” series.

Dany received his degree in the Arts from the University of Paris as well as a CPA.

Email: danyg@sgi.com

Course Syllabus

Lighting and Shading Techniques for Interactive Applications

8:30 A Introduction (Blythe)

8:35 B Lighting Model Basics (Blythe)

1. Diffuse Shading
2. Specular Highlights
3. Ambient and Emissive Lighting
4. Material Properties
5. Multi-pass Lighting
6. Directional and Positional Lights
7. Spot Lights
8. Other BRDFs
9. Global Illumination

9:10 C Shading Computations (Kilgard)

1. Per-vertex and Per-pixel Shading
2. Viewer Position and Lighting
3. Lighting with Texture Maps
   - Multi-texture
4. Light Maps
   - Diffuse
   - Specular
   - Spot Lights
5. Environment Maps
   - Sphere
   - Cube
   - Parabolic
6. Fresnel Effects

10:00 Break

10:15 D Advanced Shading I (Grantham)

1. Bump Mapping
   - Direct computation
   - Tangent-space
   - Other Methods
2. Anisotropic Reflection
3. Reflection and Refraction
   - Planar Surfaces
   - Curved Surfaces
   - Environment Maps

11:00 E Advanced Shading II (Kilgard)

1. Shadows
   - Projection
   - Shadow Volumes
   - Shadow Textures
   - Shadow Maps
   - Soft Shadows using Convolution
2. Transparency
   - Stippling
   - Blending
3. Atmospheric Effects
   - Fog
   - Depth-cuing
   - Haze
   - Non-homogeneous effects

11:45 F Summary, Questions and Answers (All)

12:00 Lunch

Course Syllabus

Advanced Graphics Programming Techniques Using OpenGL

8:30 A Introduction (Blythe)

8:35 B Visual Simulation (Blythe)

1. Tiling large Textures
2. Anisotropic Texturing
3. Developing LOD Models for Geometry
4. Billboarding
5. Light Points

9:20 C CAD I (Nelson)

1. Constructive Solid Geometry
2. Meshing and Tessellation
3. Improving Numerical Accuracy
4. Silhouettes

10:00 Break

10:15 D Graphics Special Effects (Nelson)

1. Stencil Dissolves
2. Compositing
3. Antialiasing
4. Motion Blur
5. Depth of Field

11:00 E Image Processing (McReynolds)

1. OpenGL Image Processing
2. Accum Buffer Convolution
3. Color Space Operations
4. Image Warping with Textures
5. Texture Synthesis using Noise

12:00 Lunch

1:30 F CAD II (Blythe)

1. Technical Illustration
2. Occlusion Culling Techniques
3. Depth and Transparency Cuing
4. Surface Visualization
5. Picking and Locate-highlight

2:15 G Scientific Visualization (McReynolds)

1. Scalar Field Visualization
2. Volume Rendering
3. Vector Field Visualization

3:00 Break

3:15 H Production Graphics (Blythe)

1. Character Rendering
2. Manipulating Large Images
3. 2D and Line Rendering
4. Tone-reproduction

4:00 I Simulating Natural Phenomena (Grantham)

1. Particle Systems
2. Smoke
3. Fire
4. Clouds
5. Water
6. Precipitation and Lightning
7. Fog

5:00 J Summary, Question and Answers (All)

Contents

List of Figures

List of Tables

1 Introduction

Since its first release in 1992, OpenGL has been rapidly adopted as the graphics API of choice for real-time interactive 3D graphics applications. The OpenGL state machine is easy to understand, but its simplicity and orthogonality enable a multitude of interesting effects. The goal of this course is to demonstrate how to generate more satisfying images using OpenGL. There are three general areas of discussion: generating aesthetically pleasing or realistic looking basic images, computing interesting effects, and generating more sophisticated images.

1.1 OpenGL Version

We have assumed that the attendees have a strong working knowledge of OpenGL. As much as possible we have tried to include interesting examples involving only those commands in the most readily available version of OpenGL, version 1.1, but we have not restricted ourselves to this version. At the time of this writing, the OpenGL 1.2 specification has been approved as well as the `ARB_multitexture` extension and implementations are just starting to become available. Consequently, we've used those features when it seemed sensible, but mention that we're doing so.

OpenGL is an evolving standard and we have taken the liberty of incorporating material that uses some multi-vendor extensions and, in some cases, vendor specific extensions. We do this to help make you aware of extensions that we think have general usefulness and should be more widely available.

The course notes include reprints of selected papers describing rendering techniques relevant to OpenGL, but may refer to other APIs such as OpenGL's predecessor, Silicon Graphics' IRIS GL. For new material developed for the course notes, we use terminology and notation consistent with other OpenGL documentation.

1.2 Course Notes and Slide Set Organization

For a number of reasons, these course notes do not have a one-to-one correspondence with what we present at the SIGGRAPH course. There is just too much material to present in a one-day course (in fact, we have broken out a separate half day course from the same notes to cover the large amount of material on lighting and shading alone), but we want to provide you with as much material as possible. The organization of the course presentation is constrained by presentation and time restrictions, and isn't necessarily the optimal way to organize the material. As a result, the slides and the course notes go their separate ways, and unfortunately, it is impossible to track the presenter's lectures using these notes.

We've tried to make up for this by making the slide set available on our web site, described in Section 1.6. We intend to get an accurate copy of the course materials on the web site as early as possible prior to the presentation.

1.3 Acknowledgments

This year we have tried to keep pace with a growing number of innovative techniques using OpenGL and have added a significant amount of new material as well as improvements and corrections to the old material. As usual, we've tried to do a lot in a short period of time and are grateful for the enthusiastic assistance we have received:

Some cool new ideas were contributed by Kurt Akeley, Brian Cabral, Amy Gooch, Wolfgang Heidrich, Detlev Stalling, and Hansong Zhang.

Our reviewers this year included Dave Shreiner, Paul Strauss, David Yu, and Hansong Zhang. Dany Galgani, Bob Brown, and Linda Rae Sande helped with the production. Bowen 'Cheetah' Goletz helped with the logistics of sharing the source material over the internet. We are also indebted to those who have made tools such as TeX/LaTeX, GhostScript/Ghostview, and cvs freely available on the three different computing platforms that we used for preparing the notes.

We would also like to thank John Airey, Amy Gooch, Paul Heckbert, Wolfgang Heidrich, Mark Segal, Detlev Stalling, Michael Teschner, Bruce Walter, and Tim Wiegand for providing material for inclusion in the reprints section.

Permission to reproduce [101] has been granted by Computer Graphics Forum. Permission to reproduce [92] has been granted by the IEEE.

1.4 Acknowledgments for 1998 Course Notes

Once again this year, we tried to improve the quality of our existing course notes, add a significant amount of new material, and still do our real jobs in a short amount of time. As before, we've had a lot of great help:

For still more cool ideas and demos, we'd like to thank Kurt Akeley, Luis Barcena, Brian Cabral, Angus Dorbie, Bob Drebin, Mark Peercy, Nacho Sanz-Pastor Revorio, Chris Tanner, and David Yu.

Our reviewers should also get credit for helping us fix up our mistakes: Sharon Clay, Robert Grzeszczuk, Phil Lacroute, Mark Peercy, Lena Petrovic, Allan Schaffer, and Mark Stadler.

We have a production team! Linda Rae Sande performed invaluable production editing on the entire set of course notes, improving them immensely. Dany Galgani managed to plow through nearly all of our illustrations, bringing them up to an entirely new level of quality. Chris Everett has once again helped us with the mysteries of PDF documents.

As before, we would also like to thank John Airey, Paul Heckbert, Phil Lacroute, Mark Segal, Michael Teschner, Bruce Walter, and Tim Wiegand for providing material for inclusion in the reprints section.

Permission to reproduce [101] has been granted by Computer Graphics Forum.

1.5 Acknowledgments for 1997 Course Notes

The authors have tried to compile together more than a decade worth of experience, tricks, hacks and wisdom that has often been communicated by word of mouth, code fragments or the occasional magazine or journal article. We are indebted to our colleagues at Silicon Graphics for providing us with interesting material, references, suggestions for improvement, sample programs and cool hardware.

We'd like to thank some of our more fruitful and patient sources of material: John Airey, Remi Arnaud, Brian Cabral, Bob Drebin, Phil Lacroute, Mark Peercy, and David Yu.

Credit should also be given to our army of reviewers: John Airey, Allen Akin, Brian Cabral, Tom Davis, Bob Drebin, Ben Garlick, Michael Gold, Robert Grzeszczuk, Paul Haeberli, Michael Jones, Phil Keslin, Phil Lacroute, Erik Lindholm, Mark Peercy, Mark Young, David Yu, and particularly Mark Segal for having the endurance to review for us two years in a row.

We would like to acknowledge Atul Narkhede and Rob Wheeler for coding prototype algorithms, and Chris Everett for once again providing his invaluable production expertise and assistance this year, and Dany Galgani for some really nice illustrations.

We would also like to thank John Airey, Paul Heckbert, Phil Lacroute, Mark Segal, Michael Teschner, and Tim Wiegand for providing material for inclusion in the reprints section.

Permission to reproduce [101] has been granted by Computer Graphics Forum.

1.6 Course Notes Web Site

We've created a webpage for the 1999 course and previous year's courses on SGI's OpenGL web site. It contains an HTML version of the course notes and downloadable source code for the demo programs mentioned in the text. The web address is:

`http://www.sgi.com/software/opengl/courses.html`

Additional pointers to the course web site are also available from the site `http://www.opengl.org`.

2 About OpenGL

Before getting into the intricacies of using OpenGL, we begin with a few comments about the philosophy behind the OpenGL API and some of the caveats that come with it.

OpenGL is a procedural rather than descriptive interface. In order to generate a rendering of a red sphere the programmer must specify the appropriate sequence of commands to set up the camera view and modeling transformations, draw the geometry for a sphere with a red color, etc. Other systems such as VRML [18] are descriptive; one simply specifies that a red sphere should be drawn at certain coordinates. The disadvantage of using a procedural interface is that the application must specify all of the operations in exacting detail and in the correct sequence to get the desired result. The advantage of this approach is that it allows great flexibility in the process of generating the image. The application is free to trade-off rendering speed and image quality by changing the steps through which the image is drawn. The easiest way to demonstrate the power of the procedural interface is to note that a descriptive interface can be built on top of a procedural interface, but not vice-versa. Think of OpenGL as a "graphics assembly language": the pieces of OpenGL functionality can be combined as building blocks to create innovative techniques and produce new graphics capabilities.

A second aspect of OpenGL is that the specification is not pixel exact. This means that two different OpenGL implementations are very unlikely to render exactly the same image. This allows OpenGL to be implemented across a range of hardware platforms [56]. If the specification were too exact, it would limit the kinds of hardware acceleration that could be used; limiting its usefulness as a standard. In practice, the lack of exactness need not be a burden — unless you plan to build a rendering farm from a diverse set of machines.

The lack of pixel exactness shows up even within a single implementation, in that different paths through the implementation may not generate the same set of fragments, although the specification does mandate a set of invariance rules to guarantee repeatable behavior across a variety of circumstances. A concrete example that one might encounter is an implementation that does not accelerate texture mapping operations, but accelerates all other operations. When texture mapping is enabled the fragment generation is performed on the host and as a consequence all other steps that precede texturing likely also occur on the host. This may result in either the use of different algorithms or arithmetic with different precision than that used in the hardware accelerator. In such a case, when texturing is enabled, a slightly different set of pixels in the window may be written compared to when texturing is disabled. For some of the algorithms presented in this course such variability can cause problems, so it is important to understand a little about the underlying details of the OpenGL implementation you are using.

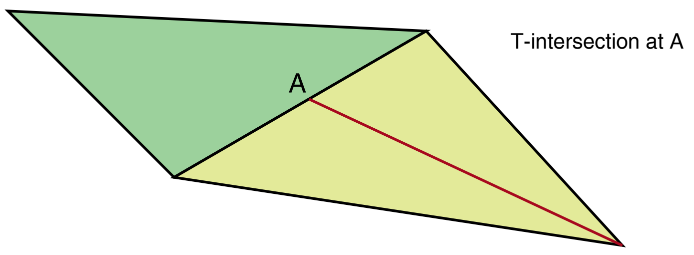


Figure 1. T-intersection

3 Modeling

Rendering is only half the story. Great computer graphics starts with great images and geometric models. This section describes some modeling rules and describes a high-performance method of performing Constructive Solid Geometry (CSG) operations.

3.1 Modeling Considerations

OpenGL is a renderer not a modeler. There are utility libraries such as the OpenGL Utility Library (GLU) that can assist with modeling tasks, but for all practical purposes modeling is the application's responsibility. Attention to modeling considerations is important; the image quality is directly related to the quality of the modeling. For example, undertessellated geometry produces poor silhouette edges. Other artifacts result from a combination of the model and OpenGL's ordering scheme. For example, interpolation of colors determined as a result of evaluation of a lighting equation at the vertices can result in a less than pleasing specular highlight if the geometry is not sufficiently sampled. We include a short list of modeling considerations with which OpenGL programmers should be familiar:

- Consider using triangles, triangle strips and triangle fans. Primitives such as polygons and quads are usually decomposed by OpenGL into triangles before rasterization. OpenGL does not provide controls over how this decomposition is done, so for more predictable results, the application should do the tessellation directly. Application tessellation is also more efficient if the same model is to be drawn multiple times (e.g., multiple instances per frame, as part of a multipass algorithm, or for multiple frames). The second release of the GLU library (version 1.1) includes a very good general polygon tessellator; it is highly recommended.
- Avoid T-intersections (also called T-vertices). T-intersections occur when one or more triangles share (or attempt to share) a partial edge with another triangle (Figure 1).

  Even though the geometry may be perfectly aligned when defined, after transformation it is no longer guaranteed to be an exact match. Since finite-precision algorithms are used to rasterize triangles, the edges will not always be perfectly aligned when they are drawn unless both edges share common vertices. This problem typically manifests itself during animations when the model is moved and cracks along the polygon edges appear and disappear. In order to avoid the problem, shared edges should share the same vertex positions so that the edge equations are the same.

  Note that this requirement must be satisfied when seemingly separate models are sharing an edge. For example, an application may have modeled the walls and ceiling of the interior of a room independently, but they *do* share common edges where they meet. In order to avoid cracking when the room is rendered from different viewpoints, the walls and ceilings should use the same vertex coordinates for any triangles along the shared edges. This often requires adding edges and creating new triangles to "stitch" the edges of abutting objects together seamlessly.

- The T-intersection problem has consequences for view-dependent tessellation. Imagine drawing an object in extreme perspective so that some part of the object maps to a large part of the screen and an equally large part of the object (in object coordinates) maps to a small portion of the screen. To minimize the rendering time for this object, applications tessellate the object to varying degrees depending on the area of the screen that it covers. This ensures that time is not wasted drawing many triangles that cover only a few pixels on the screen. This is a difficult mechanism to implement correctly; if the view of the object is changing, the changes in tessellation from frame to frame may result in noticeable motion artifacts. Often it is best to either undertessellate and live with those artifacts or overtessellate and accept reduced performance. The GLU NURBS library is an example of a package that implements view-dependent tessellation and provides substantial control over the sampling method and tolerances for the tessellation.

- Another problem related to T-intersections occurs with careless specification of surface boundaries. If a surface is intended to be closed, it should share the same vertex coordinates where the surface specification starts and ends. A simple example of this would be drawing a sphere by subdividing the interval $[0, 2\pi]$ to generate the vertex coordinates. The vertex at 0 must be the same as the one at 2π. Note that the OpenGL specification is very strict in this regard as even the `glMapGrid` routine must evaluate exactly at the boundaries to ensure that evaluated surfaces can be properly stitched together.

- Another consideration is the quality of the attributes that are specified with the vertex coordinates, in particular, the vertex (or face) normals and texture coordinates. When computing normals for an object, sharp edges should have separate normals at common vertices, while smooth edges should have common normals. For example, a cube is made up of six quadrilaterals where each vertex is shared by three polygons, but a different normal should be used for each of the three instances of each vertex, but a sphere is made up of many polygons where all vertices have common normals. Failure to properly set these attributes can result in unnatural lighting effects or shading techniques such as environment mapping will exaggerate the errors resulting in unacceptable artifacts.

- The final suggestion is to be consistent about the orientation of polygons. That is, ensure that all polygons on a surface are oriented in the same direction (clockwise or counterclockwise) when viewed from the outside. The OpenGL face culling method is an efficient form of hidden surface elimination for closed surfaces.

3.2 Decomposition and Tessellation

Tessellation refers to the process of decomposing a complex surface such as a sphere into simpler primitives such as triangles or quadrilaterals. Most OpenGL implementations are tuned to process triangle strips and triangle fans efficiently. Triangles are desirable because they are planar, easy to rasterize, and can always be interpolated unambiguously. When an implementation is optimized for processing triangles, more complex primitives such as quad strips, quads, and polygons are decomposed into triangles early in the pipeline.

If the underlying implementation is performing this decomposition, there is a performance benefit in performing this decomposition *a priori,* either when the database is created or at application initialization time, rather than each time the primitive is issued. A second advantage of performing this decomposition under the control of the application is that the decomposition can be done consistently and independently of the OpenGL implementation. Since OpenGL does not specify its decomposition algorithm, different implementations may decompose a given quadrilateral along different diagonals. This can result in an image that is shaded differently and has different silhouette edges when drawn on two different OpenGL implementations.

Quadrilaterals may be decomposed by finding the diagonal that creates two triangles with the least difference in orientation. A good way to find this diagonal is to compute the angles between the normals at opposing vertices, compute the dot product, then choose the pair with the smallest angle (largest dot product) as shown in Figure 2. The normals for a vertex can be computed by taking the cross products of the the two vectors with origins at that

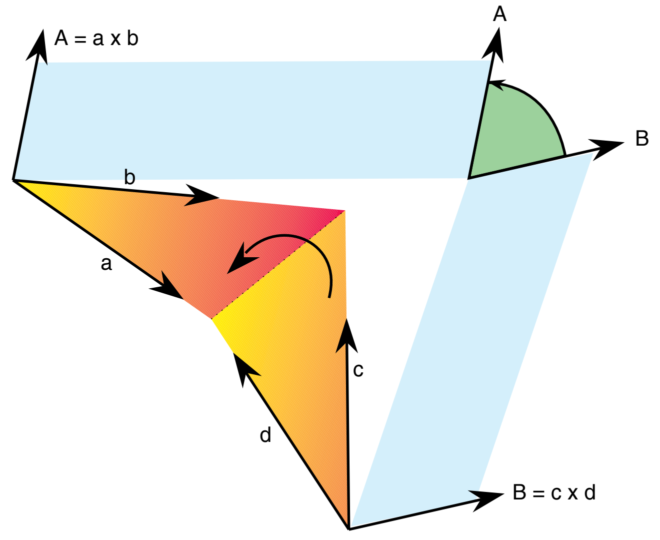


Figure 2. Quadrilateral Decomposition

vertex. An alternative decomposition method is to split the quadrilateral into triangles that are closest to equal in size.

Tessellation of simple surfaces such as spheres and cylinders is not difficult. Most implementations of the GLU library use a simple latitude-longitude tessellation for a sphere. While the algorithm is simple to implement, it has the disadvantage that the triangles produced from the tessellation have widely varying sizes. These widely varying sizes can cause noticeable artifacts, particularly if the object is lit and rotating.

A better algorithm generates triangles with sizes that are more consistent. Octahedral and icosahedral tessellations work well and are not very difficult to implement. An octahedral tessellation approximates a sphere with an octahedron whose vertices are all on the unit sphere. Since the faces of the octahedron are triangles they can easily be split into four triangles, as shown in Figure 3.

Each triangle is split by creating a new vertex in the middle of each edge and adding three new edges. These vertices are scaled onto the unit sphere by dividing them by their distance from the origin (normalizing them). This process can be repeated as desired, recursively dividing all of the triangles generated in each iteration.

The same algorithm can be applied using an icosahedron as the base object, recursively dividing all 20 sides. In both cases the algorithms can be coded so that triangle strips are generated instead of independent triangles, maximizing rendering performance. It is not necessary to split the triangle edges in half, since tessellating the triangle by other amounts, such as by thirds, or even any arbitrary number, may produce a more desirable final uniform triangle size.

3.3 Generating Model Normals

Given an arbitrary polygonal model without precomputed normals, it is fairly easy to generate polygon normals for faceted shading, but quite a bit more difficult to create correct vertex normals for smooth shading. A simple cross product of two edges followed by a normalization of the result to obtain a unit-length vector generates a facet normal. Computing a correct vertex normal must take into account all facets that share that normal and whether or not all facets should contribute to the normal. For best results, compute all normals before converting to triangle

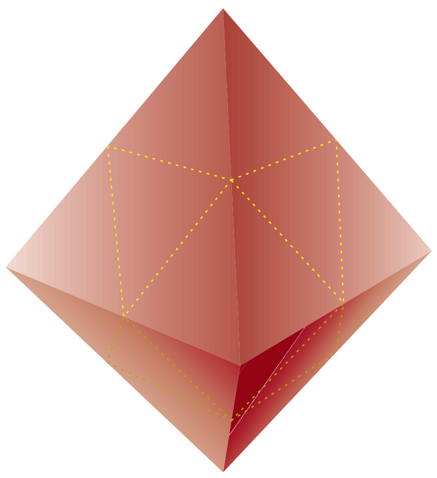


Figure 3. Octahedron with Triangle Subdivision

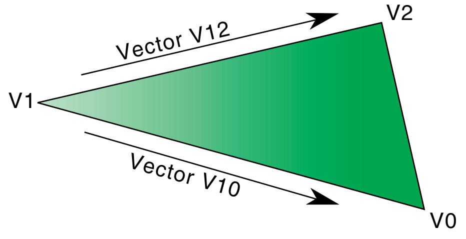


Figure 4. Computing a Surface Normal from Edges' Cross Product

strips.

To compute the facet normal of a triangle, select one vertex, compute the vectors from that vertex to the other two vertices, then compute the cross product of those two vectors. Figure 4 shows which vectors to use to compute a cross product for a triangle. The following code fragment generates a facet normal for a triangle, assuming a clockwise polygon winding when viewed from the front:

```
/* Compute edge vectors */
x10 = x1 - x0;
y10 = y1 - y0;
z10 = z1 - z0;
x12 = x1 - x2;
y12 = y1 - y2;
z12 = z1 - z2;

/* Compute the cross product */
cpx = (z10 * y12) - (y10 * z12);
cpy = (x10 * z12) - (z10 * x12);
cpz = (y10 * x12) - (x10 * y12);
```

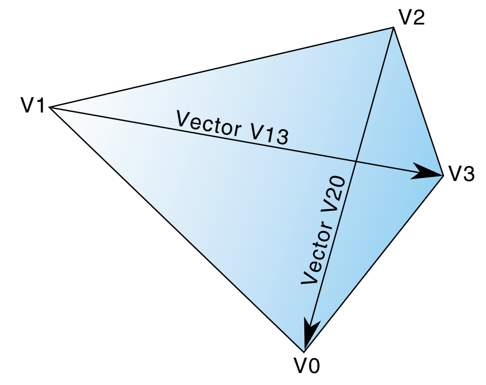


Figure 5. Computing Quadrilateral Surface Normal from Vertex Cross Product

```
    /* Normalize the result to get the unit-length facet normal */
    r = sqrt(cpx * cpx + cpy * cpy + cpz * cpz);
    nx = cpx / r;
    ny = cpy / r;
    nz = cpz / r;
```

Computing the facet normal of a polygon with more than three vertices is a bit trickier. Often such polygons are not perfectly planar, so you may get a different result depending on which three vertices are chosen. If the polygon is a quadrilateral one good method is to take the cross product of the vectors between opposing vertices as shown in Figure 5. The following code fragment computes the cross product for a quadrilateral:

```
    /* Compute vectors */
    x20 = x2 - x0;
    y20 = y2 - y0;
    z20 = z2 - z0;
    x13 = x1 - x3;
    y13 = y1 - y3;
    z13 = z1 - z3;

    /* Compute the cross product */
    cpx = (z20 * y13) - (y20 * z13);
    cpy = (x20 * z13) - (z20 * x13);
    cpz = (y20 * x13) - (x20 * y13);
```

For polygons with more than four vertices it can be difficult to choose the best vertices to use for computing the cross product. It is best to attempt to choose vertices that are the furthest apart from each other, if possible, or average the result of several vertex cross products.

3.3.1 Consistent Vertex Winding

Some 3D models come with polygons that are not all wound in a clockwise or counterclockwise direction, but are a mixture of both. Those polygons that are wound inconsistently should have the vertex order reversed. A good

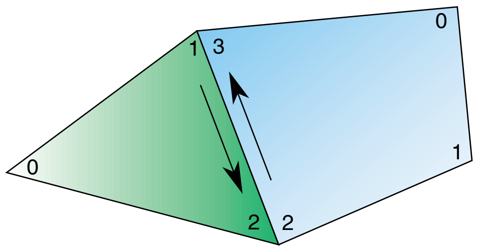


Figure 6. Proper Winding for Shared Edge of Adjoining Facets

way to accomplish this is to find all common edges and verify that neighboring polygon edges are drawn in the opposite order (see Figure 6).

To begin rewinding polygons, one polygon must be chosen as “correct.” All neighboring polygons must then be found and made consistent with the “correct” polygon. This repeats recursively for each new “correct” polygon until no more neighboring polygons can be found. If the model is a single closed object, all polygons will now be consistent. However, if the model has multiple unconnected pieces, another polygon that has not yet been tested must be found and the process must be repeated until all polygons have been tested and made consistent.

The above method still leaves a 50-50 chance that the entire object is now wound backwards (assuming an object with half of the facets wound clockwise and half wound counterclockwise). Short of getting a human involved to look at the model, there are ways to check that the normals are pointing outwards. One way is to find the geometric center of the object by computing the object bounding box by finding the maximum and minimum X, Y and Z values, then computing the mid-point of the bounding box. Next, select a vertex that is the maximum distance from this center point and compute the (normalized) vector from the center point to this vertex. Then take the normal of one of the facets that shares the distant vertex and compute the dot product of the two vectors. A positive result indicates that the normals are all correct while a negative result indicates that the normals are all backwards. If the normals are backwards, negate them all and reverse the windings of all facets.

There are still a few pathological cases that may not come out right, such as a model of a room where it is desirable to view the inside walls, but the above method works for most cases.

3.3.2 Smooth Shading

To smoothly shade an object, the same normal should be used on a given vertex for all polygons that share the vertex. The simplest way to do this is to add all (normalized) normals from the common facets then renormalize the result [38]. This provides reasonable results for surfaces that are fairly smooth, but does not look good for surfaces with sharp edges.

An object with a sharp corner, such as a cube, should look like it has a hard edge, rather than a soft edge. The angle between polygons that should produce a hard edge can vary from model to model. It is fairly clear that a 90 degree edge should always be considered a hard edge, but some models look better with hard edges at angles less than 45 degrees while others look better with soft edges for angles greater than 45 degrees. This particular parameter should generally be left under user control with a good default probably right around 45 degrees.

To determine the angle between polygons, take the dot product of the facet normals (which must be unit length). A dot product returns the cosine of the angle between the vectors. So, if the dot product of the two normals is greater than the cosine of the desired hard edge angle, the edge should be considered soft, otherwise it should be considered hard. To create a hard edge, a different normal is generated for each side. Be sure to keep common normals for any remaining soft edges of the surface.

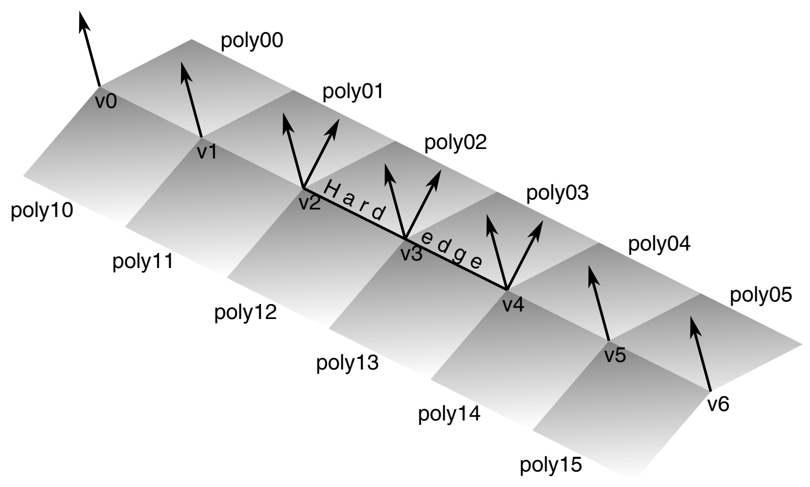


Figure 7. Splitting Normals for Hard Edges

Figure 7 shows an example of a mesh with two hard edges in it. The three vertices making up these hard edges, *v2*, *v3*, and *v4*, need to be split using two separate normals. In the case of vertex *v2*, one normal would apply to *poly01* and *poly02* and a different normal would apply to *poly11* and *poly12*. This makes sure that the edge between *poly01* and *poly02* still looks smooth while the edge between *poly02* and *poly12* has a nice crease and looks like a sharp edge. Since *v1* is not split, the edge between *poly01* and *poly11* will look sharper near *v2* and will become smoother as it gets closer to *v1*. The edge between *v1* and *v0* would then be completely smooth. This is the desired effect.

For an object such as a cube, three hard edges will share one common vertex. In this case the edge splitting algorithm needs to be repeated for the third edge to achieve the correct results.

3.4 Triangle-stripping

One of the simplest ways to speed up an OpenGL program while simultaneously saving storage space is to convert independent triangles or polygons into triangle strips. If the model is generated directly from NURBS data or from some other regular geometry, it is quite straightforward to connect the triangles together into longer strips. You must keep in mind whether you want the first triangle to start off with a clockwise or counterclockwise winding, then all subsequent triangles in the list will alternate winding (see Figure 8). Triangle fans must also be started with the correct winding, but all subsequent triangles are wound in the same direction (see Figure 9).

Because OpenGL does not have a way to specify generalized triangle strips, the user must choose between `GL_TRIANGLE_STRIP` and `GL_TRIANGLE_FAN`. In general, more triangles can be placed into a strip than a fan. Triangle fans are great when a large convex polygon needs to be converted to triangles or for geometry that is cone-shaped. Most other cases are best converted to triangle strips.

For regular meshes, triangle strips should be lined up side by side as shown in Figure 10. The goal here is to minimize the number of total strips and try to avoid "orphan" triangles (also known as *singleton strips*) that ca not be made part of a longer strip. It is possible to turn a corner in a triangle strip by using redundant vertices and degenerate triangles as described in [26].

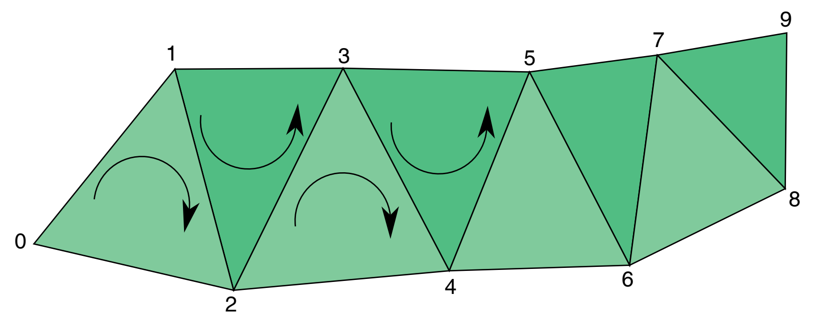


Figure 8. Triangle Strip Winding

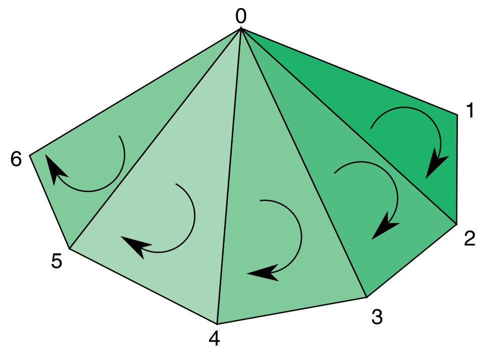


Figure 9. Triangle Fan Winding

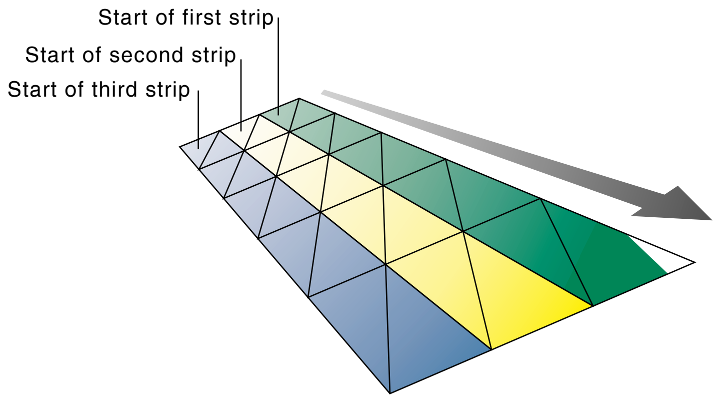


Figure 10. A Mesh Made up of Multiple Triangle Strips

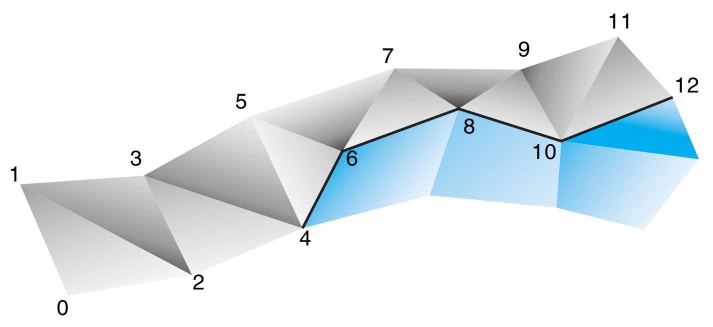


Figure 11. "Greedy" Triangle Strip Generation

3.4.1 Greedy Tri-stripping

A fairly simple method of converting a model into triangle strips is sometimes known as greedy tri-stripping. One of the early greedy algorithms was developed for IRIS GL that allowed swapping of vertices to create direction changes to the facet with the least neighbors. However, with OpenGL the only way to get the equivalent behavior of swapping vertices is to repeat a vertex and create a degenerate triangle, which is more expensive than the original vertex swap operation.

For OpenGL a better algorithm is to choose a polygon, convert it to triangles, then continue onto the neighboring polygon from the last edge of the previous polygon. For a given starting polygon beginning at a given edge, there are no choices as to which polygon is the best to choose next since there is only one choice. The strip is continued until the triangle strip runs off the edge of the model or runs into a polygon that is already a part of another strip (see Figure 11). For best results, pick a polygon and go both directions as far as possible, then start the triangle strip from one end.

A triangle strip should not cross a hard edge, unless the vertices on that edge are repeated redundantly, since you will want different normals for the two triangles on either side of that edge. Once one strip is complete, the best polygon to choose for the next strip is often a neighbor to the polygon at one end or the other of the previous strip. More advanced triangulation methods do not try to keep all triangles of a polygon together. For more information on such a method refer to [26].

3.5 Coplanar Polygons and Decaling with Stencil

Using stenciling to control pixels drawn from a particular primitive can help solve a number of important problems:

1. Drawing depth-buffered, co-planar polygons without z-buffering artifacts.
2. Decaling multiple textures on a primitive.

Values are written to the stencil buffer to create a mask for area to be decaled. Then this stencil mask is used to control two separate draw steps; one for the decaled region, one for the rest of the polygon.

A useful example that illustrates the technique is rendering co-planar polygons. If one polygon is to be rendered directly on top of another (runway markings, for example), the depth buffer ca not be relied upon to produce a clean separation between the two. This is due to the quantization of the depth buffer. Since the polygons have different vertices, the rendering algorithms can produce z values that are rounded to the wrong depth buffer value, so some pixels of the back polygon may show through the front polygon. In an application with a high frame rate,

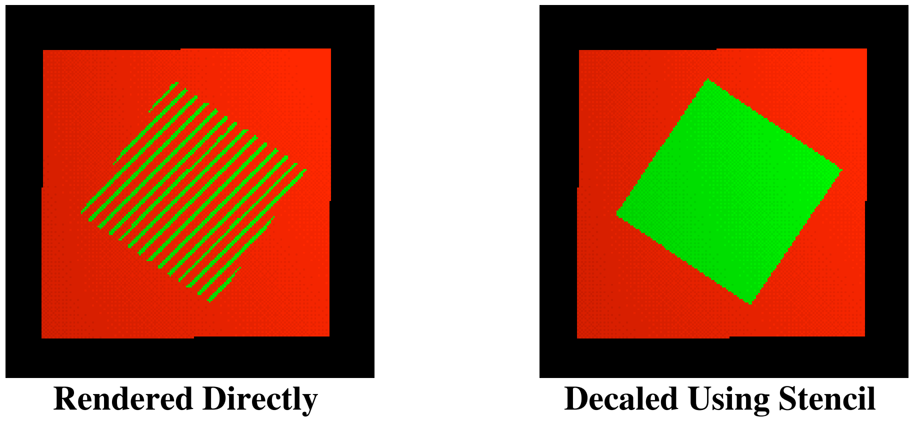


Figure 12. Using Stencil to Render Co-planar Polygons

this results in a shimmering mixture of pixels from both polygons (commonly called "Z fighting" or "flimmering"). An example is shown in in Figure 12.

To solve this problem, the closer polygons are drawn with the depth test disabled, on the same pixels covered by the farthest polygons. It appears that the closer polygons are "decaled" on the farther polygons.

Decaled polygons can be drawn with the following steps:

1. Turn on stenciling; `glEnable(GL_STENCIL_TEST)`.
2. Set stencil function to always pass; `glStencilFunc(GL_ALWAYS, 1, 1)`.
3. Set stencil op to set 1 if depth passes, 0 if it fails; `glStencilOp(GL_KEEP, GL_ZERO, GL_REPLACE)`.
4. Draw the base polygon.
5. Set stencil function to pass when stencil is 1; `glStencilFunc(GL_EQUAL, 1, 1)`.
6. Disable writes to stencil buffer; `glStencilMask(GL_FALSE)`.
7. Turn off depth buffering; `glDisable(GL_DEPTH_TEST)`.
8. Render the decal polygon.

The stencil buffer does not have to be cleared to an initial value; the stencil values are initialized as a side effect of writing the base polygon. Stencil values will be one where the base polygon was successfully written into the framebuffer, and zero where the base polygon generated fragments that failed the depth test. The stencil buffer becomes a mask, ensuring that the decal polygon can only affect the pixels that were touched by the base polygon. This is important if there are other primitives partially obscuring the base polygon and decal polygons.

There are a few limitations to this technique. First, it assumes that the decal polygon does not extend beyond the edge of the base polygon. If it does, you will have to clear the entire stencil buffer before drawing the base polygon, which is expensive on some machines. If you are careful to redraw the base polygon with the stencil operations set to zero the stencil after you've drawn each decaled polygon, you will only have to clear the entire stencil buffer once, for any number of decaled polygons.

Second, if the screen extents of the base polygons you're decaling overlap, you will have to perform the decal process for one base polygon and its decals before you move on to another base and decals. This is an important consideration if your application collects and then sorts geometry based on its graphics state, where the rendering order of geometry may be changed by the sort.

This process can be extended to allow a number of overlapping decal polygons, the number of decals limited by the number of stencil bits available for the visual. The decals do not have to be sorted. The procedure is the similar to the previous algorithm, with the following extensions.

Assign a stencil bit for each decal and the base polygon. The lower the number, the higher the priority of the polygon. Render the base polygon as before, except instead of setting its stencil value to one, set it to the largest priority number. For example, if there were three decal layers, the base polygon would have a value of 8.

When you render a decal polygon, only draw it if the decal's priority number is lower than the pixels it is trying to change. For example, if the decal's priority number was 1, it would be able to draw over every other decal and the base polygon; `glStencilFunc(GL_LESS, 1, 0)` and `glStencilOp(GL_KEEP, GL_REPLACE, GL_REPLACE)`.

Decals with the lower priority numbers will be drawn on top of decals with higher ones. Since the region not covered by the base polygon is zero, no decals can write to it. You can draw multiple decals at the same priority level. If you overlap them, however, the last one drawn will overlap the previous ones at the same priority level.

Multiple textures can be drawn onto a polygon with a similar technique. Instead of writing decal polygons, the same polygon is drawn with each subsequent texture and an alpha value to blend the old pixel color and the new pixel color together.

3.6 Capping Clipped Solids with the Stencil Buffer

When dealing with solid objects it is often useful to clip the object against a plane and observe the cross section. OpenGL's user-defined clipping planes allow an application to clip the scene by a plane. The stencil buffer provides an easy method for adding a "cap" to objects that are intersected by the clipping plane. A capping polygon is embedded in the clipping plane and the stencil buffer is used to trim the polygon to the interior of the solid.

The stencil buffer is described in more detail in Section 8.6.

If some care is taken when constructing the object, solids that have a depth complexity greater than 2 (concave or shelled objects) and less than the maximum value of the stencil buffer can be rendered. Object surface polygons must have their vertices ordered so that they face away from the interior for face culling purposes.

The stencil buffer, color buffer, and depth buffer are cleared, and color buffer writes are disabled. The capping polygon is rendered into the depth buffer, then depth buffer writes are disabled. The stencil operation is set to increment the stencil value where the depth test passes, and the model is drawn with `glCullFace(GL_BACK)`. The stencil operation is then set to decrement the stencil value where the depth test passes, and the model is drawn with `glCullFace(GL_FRONT)`.

At this point, the stencil buffer is 1 wherever the clipping plane is enclosed by the frontfacing and backfacing surfaces of the object. The depth buffer is cleared, color buffer writes are enabled, and the polygon representing the clipping plane is now drawn using whatever material properties are desired, with the stencil function set to `GL_EQUAL` and the reference value set to 1. This draws the color and depth values of the cap into the framebuffer only where the stencil values equal 1.

Finally, stenciling is disabled, the OpenGL clipping plane is applied, and the clipped object is drawn with color and depth enabled.

3.7 Constructive Solid Geometry with the Stencil Buffer

Before continuing, the it may help for the reader to be familiar with the concepts of stencil buffer usage presented in Section 8.6.

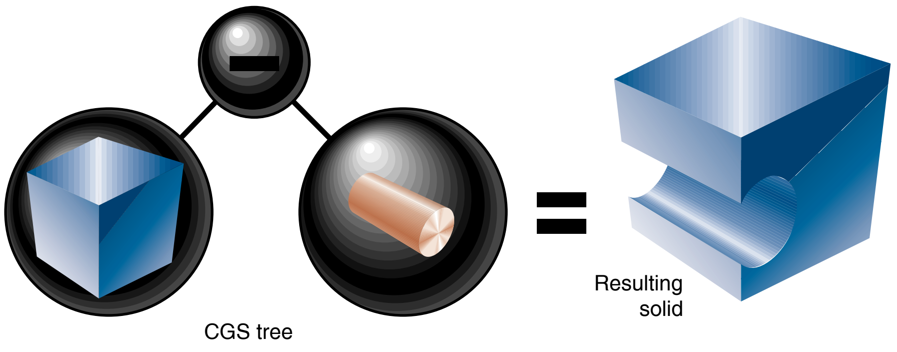


Figure 13. An Example Of Constructive Solid Geometry

Constructive solid geometry (CSG) models are constructed through the intersection ($\cap$), union ($\cup$), and subtraction ($-$) of solid objects, some of which may be CSG objects themselves[33]. The tree formed by the binary CSG operators and their operands is known as the CSG tree. Figure 13 shows an example of a CSG tree and the resulting model.

The representation used in CSG for solid objects varies, but we will consider a solid to be a collection of polygons forming a closed volume. "Solid," "primitive," and "object" are used here to mean the same thing.

CSG objects have traditionally been rendered through the use of ray-casting, which is slow, or through the construction of a boundary representation (B-rep).

B-reps vary in construction, but are generally defined as a set of polygons that form the surface of the result of the CSG tree. One method of generating a B-rep is to take the polygons forming the surface of each primitive and trim away the polygons (or portions thereof) that do not satisfy the CSG operations. B-rep models are typically generated once and then manipulated as a static model because they are slow to generate.

Drawing a CSG model using stencil usually means drawing more polygons than a B-rep would contain for the same model. Enabling stencil also may reduce performance. Nonetheless, some portions of a CSG tree may be interactively manipulated using stencil if the remainder of the tree is cached as a B-rep.

The algorithm presented here is from a paper by Tim F. Wiegand describing a GL-independent method for using stencil in a CSG modeling system for fast interactive updates. The technique can also process concave solids, the complexity of which is limited by the number of stencil planes available. A reprint of Wiegand's paper is included in the Appendix.

The algorithm presented here assumes that the CSG tree is in "normal" form. A tree is in normal form when all intersection and subtraction operators have a left subtree that contains no union operators and a right subtree that is simply a primitive (a set of polygons representing a single solid object). All union operators are pushed towards the root, and all intersection and subtraction operators are pushed towards the leaves. For example, $(((A \cap B) - C) \cup (((D \cap E) \cap G) - F)) \cup H$ is in normal form; Figure 14 illustrates the structure of that tree and the characteristics of a tree in normal form.

A CSG tree can be converted to normal form by repeatedly applying the following set of production rules to the tree and then its subtrees:

1. $X - (Y \cup Z) \rightarrow (X - Y) - Z$
2. $X \cap (Y \cup Z) \rightarrow (X \cap Y) \cup (X \cap Z)$
3. $X - (Y \cap Z) \rightarrow (X - Y) \cup (X - Z)$

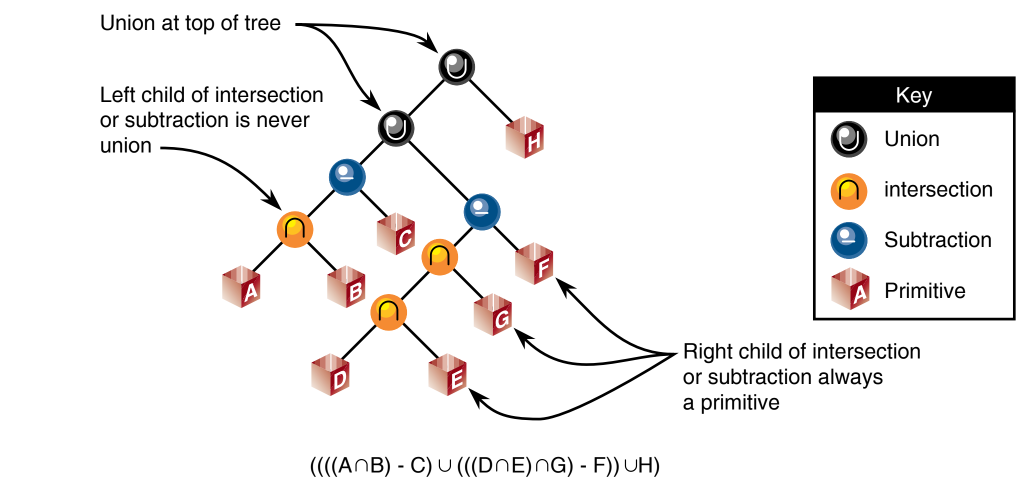

Union at top of tree
Left child of intersection or subtraction is never union
Key
Union
intersection
Subtraction
Primitive
A
B
C
D
E
F
G
H
Right child of intersection or subtraction always a primitive
((((A∩B) - C) ∪ (((D∩E)∩G) - F)) ∪H)

Figure 14. A CSG Tree in Normal Form

4. $X \cap (Y \cap Z) \rightarrow (X \cap Y) \cap Z$
5. $X - (Y - Z) \rightarrow (X - Y) \cup (X \cap Z)$
6. $X \cap (Y - Z) \rightarrow (X \cap Y) - Z$
7. $(X - Y) \cap Z \rightarrow (X \cap Z) - Y$
8. $(X \cup Y) - Z \rightarrow (X - Z) \cup (Y - Z)$
9. $(X \cup Y) \cap Z \rightarrow (X \cap Z) \cup (Y \cap Z)$

X, Y, and Z here match either primitives or subtrees. Here is the algorithm used to apply the production rules to the CSG tree:

```
normalize(tree *t)
{
    if (isPrimitive(t))
        return;

    do {
        while (matchesRule(t))  /* Using rules given above */
            applyFirstMatchingRule(t);
        normalize(t->left);
    } while (!(isUnionOperation(t) ||
         (isPrimitive(t->right) &&
          ! isUnionOperation(T->left))));
    normalize(t->right);
}
```

Normalization may increase the size of the tree and add primitives that do not contribute to the final image. The bounding volume of each CSG subtree can be used to prune the tree as it is normalized. Bounding volumes for the tree may be calculated using the following algorithm:

```
findBounds(tree *t)
{
    if (isPrimitive(t))
        return;

    findBounds(t->left);
    findBounds(t->right);

    switch (t->operation){
      case UNION:
        t->bounds = unionOfBounds(t->left->bounds,
                                  t->right->bounds);
      case INTERSECTION:
        t->bounds = intersectionOfBounds(t->left->bounds,
                                         t->right->bounds);
      case SUBTRACTION:
        t->bounds = t->left->bounds;
    }
}
```

CSG subtrees rooted by the intersection or subtraction operators may be pruned at each step in the normalization process using the following two rules:

1. If `T` is an intersection and not `intersects(T->left->bounds, T->right->bounds)`, delete `T`.
2. If `T` is a subtraction and not `intersects(T->left->bounds, T->right->bounds)`, replace `T` with `T->left`.

The normalized CSG tree is a binary tree, but it is important to think of the tree rather as a "sum of products" to understand the stencil CSG procedure.

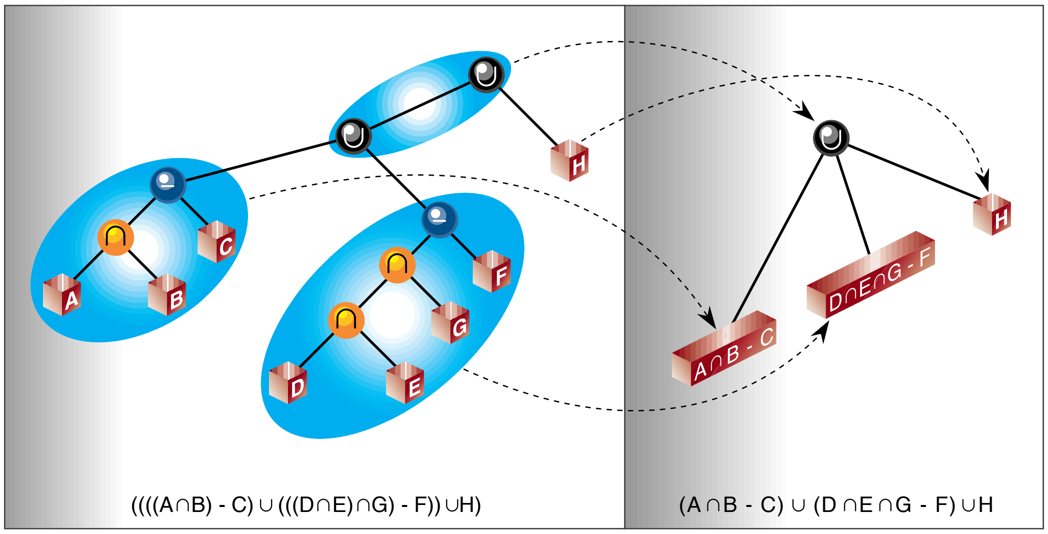
Consider all the unions as sums. Next, consider all the intersections and subtractions as products. (Subtraction is equivalent to intersection with the complement of the term to the right. For example, $A - B = A \cap \bar{B}$.) Imagine all the unions flattened out into a single union with multiple children; that union is the "sum." The resulting subtrees of that union are all composed of subtractions and intersections, the right branch of those operations is always a single primitive, and the left branch is another operation or a single primitive. You should read each child subtree of the imaginary multiple union as a single expression containing all the intersection and subtraction operations concatenated from the bottom up. These expressions are the "products." For example, you should think of $((A \cap B) - C) \cup (((G \cap D) - E) \cap F) \cup H$ as meaning $(A \cap B - C) \cup (G \cap D - E \cap F) \cup H$. Figure 15 illustrates this process.

At this time, redundant terms can be removed from each product. Where a term subtracts itself ($A - A$), the entire product can be deleted. Where a term intersects itself ($A \cap A$), that intersection operation can be replaced with the term itself.

All unions can be rendered simply by finding the visible surfaces of the left and right subtrees and letting the depth test determine the visible surface. All products can be rendered by drawing the visible surfaces of each primitive in the product and trimming those surfaces with the volumes of the other primitives in the product. For example, to render $A - B$, the visible surfaces of A are trimmed by the complement of the volume of B, and the visible surfaces of B are trimmed by the volume of A.

The visible surfaces of a product are the front facing surfaces of the operands of intersections and the back facing surfaces of the right operands of subtraction. For example, in $(A - B \cap C)$, the visible surfaces are the front facing surfaces of A and C, and the back facing surfaces of B.

Concave solids are processed as sets of front or back facing surfaces. The "convexity" of a solid is defined as the maximum number of pairs of front and back surfaces that can be drawn from the viewing direction. Figure 16



A
B
C
D
E
F
G
H
A ∩ B - C
D∩E∩G - F
((((A∩B) - C) ∪ (((D∩E)∩G) - F)) ∪H)
(A ∩B - C) ∪ (D ∩E ∩G - F) ∪H

Figure 15. Thinking of a CSG Tree as a Sum of Products

shows some examples of the convexity of objects. The *n*th front surface of a *k*-convex primitive is denoted A_{nf}, and the *n*th back surface is A_{nb}. Because a solid may vary in convexity when viewed from different directions, accurately representing the convexity of a primitive may be difficult and may also involve reevaluating the CSG tree at each new view. Instead, the algorithm must be given the *maximum possible* convexity of a primitive, and draws the *n*th visible surface by using a counter in the stencil planes.

The CSG tree must be further reduced to a "sum of partial products" by converting each product to a union of products, each consisting of the product of the visible surfaces of the target primitive with the remaining terms in the product.

For example, if A, B, and D are 1-convex and C is 2-convex:

$$\begin{array}{rl}
(A - B \cap C \cap D) & \rightarrow \\
(A_{0f} - B \cap C \cap D) & \cup \\
(B_{0b} \cap A \cap C \cap D) & \cup \\
(C_{0f} \cap A - B \cap D) & \cup \\
(C_{1f} \cap A - B \cap D) & \cup \\
(D_{0f} \cap A \cap B \cap C) &
\end{array}$$

Because the target term in each product has been reduced to a single front or back facing surface, the bounding volumes of that term will be a subset of the bounding volume of the original complete primitive. Once the tree is converted to partial products, the pruning process may be applied again with these subset volumes.

In each resulting child subtree representing a partial product, the leftmost term is called the "target" surface, and the remaining terms on the right branches are called "trimming" primitives.

The resulting sum of partial products reduces the rendering problem to rendering each partial product correctly before drawing the union of the results. Each partial product is rendered by drawing the target surface of the partial product and then "classifying" the pixels generated by that surface with the depth values generated by each of the

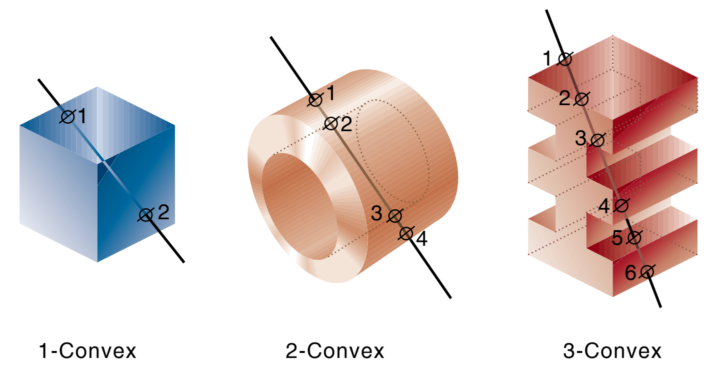


Figure 16. Examples of *n*-convex Solids

trimming primitives in the partial product. If pixels drawn by the trimming primitives pass the depth test an even number of times, that pixel in the target primitive is "out," and discarded. If the count is odd, the target primitive pixel is "in," and kept.

Because the algorithm saves depth buffer contents between each object, we optimize for depth saves and restores by drawing as many of target and trimming primitives for each pass as we can fit in the stencil buffer.

The algorithm uses one stencil bit (S_p) as a toggle for trimming primitive depth test passes (parity), n stencil bits for counting to the nth surface (S_{count}), where n is the smallest number for which 2^n is larger than the maximum convexity of a current object, and as many bits are available (S_a) to accumulate whether target pixels have to be discarded. Because S_{count} will require the GL_INCR operation, it must be stored contiguously in the least-significant bits of the stencil buffer. S_p and S_{count} are used in two separate steps, and so may share stencil bits.

For example, drawing two 5-convex primitives requires one S_p bit, three S_{count} bits, and two S_a bits. Because S_p and S_{count} are independent, the total number of stencil bits required is 5.

Once the tree is converted to a sum of partial products, the individual products are rendered. Products are grouped together so that as many partial products can be rendered between depth buffer saves and restores as the stencil buffer has capacity.

For each group, writes to the color buffer are disabled, the contents of the depth buffer are saved, and the depth buffer is cleared. Then, every target in the group is classified against its trimming primitives. The depth buffer is then restored, and every target in the group is rendered against the trimming mask. The depth buffer save/restore can be optimized by saving and restoring only the region containing the screen-projected bounding volumes of the target surfaces.

```
for each group
    glReadPixels(...);
    <classify the group>
    glStencilMask(0);   /* so DrawPixels won't affect Stencil */
    glDrawPixels(...);
    <render the group>
```

Classification consists of drawing each target primitive's depth value and then clearing those depth values where the target primitive is determined to be outside the trimming primitives.

```
glClearDepth(far);
```

```
glClear(GL_DEPTH_BUFFER_BIT);
a = 0;
for (each target surface in the group)
    for (each partial product targeting that surface)
        <render the depth values for the surface>
        for (each trimming primitive in that partial product)
            <trim the depth values against that primitive>
        <set Sa to 1 where Sa = 0 and Z < Zfar>
        a++;
```

The depth values for the surface are rendered by drawing the primitive containing the the target surface with color and stencil writes disabled. (S_{count}) is used to mask out all but the target surface. In practice, most CSG primitives are convex, so the algorithm is optimized for that case.

```
if (the target surface is front facing)
    glCullFace(GL_BACK);
else
    glCullFace(GL_FRONT);

if (the surface is 1-convex)
    glDepthMask(1);
    glColorMask(0, 0, 0, 0);
    glStencilMask(0);
    <draw the primitive containing the target surface>
else
    glDepthMask(1);
    glColorMask(0, 0, 0, 0);
    glStencilMask(Scount);
    glStencilFunc(GL_EQUAL, index of surface, Scount);
    glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);
    <draw the primitive containing the target surface>
    glClearStencil(0);
    glClear(GL_STENCIL_BUFFER_BIT);
```

Then each trimming primitive for that target surface is drawn in turn. Depth testing is enabled and writes to the depth buffer are disabled. Stencil operations are masked to S_p and the S_p bit in the stencil is cleared to 0. The stencil function and operation are set so that S_p is toggled every time the depth test for a fragment from the trimming primitive succeeds. After drawing the trimming primitive, if this bit is 0 for uncomplemented primitives (or 1 for complemented primitives), the target pixel is "out," and must be marked "discard," by enabling writes to the depth buffer and storing the far depth value (Z_f) into the depth buffer everywhere that the S_p indicates "discard."

```
glDepthMask(0);
glColorMask(0, 0, 0, 0);
glStencilMask(mask for Sp);
glClearStencil(0);
glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS, 0, 0);
glStencilOp(GL_KEEP, GL_KEEP, GL_INVERT);
<draw the trimming primitive>
glDepthMask(1);
```

Once all the trimming primitives are rendered, the values in the depth buffer are Z_f for all target pixels classified as "out." The S_a bit for that primitive is set to 1 everywhere that the depth value for a pixel is not equal to Z_f, and 0 otherwise.

Each target primitive in the group is finally rendered into the framebuffer with depth testing and depth writes enabled, the color buffer enabled, and the stencil function and operation set to write depth and color only where the depth test succeeds and S_a is 1. Only the pixels inside the volumes of all the trimming primitives are drawn.

```
glDepthMask(1);
glColorMask(1, 1, 1, 1);
a = 0;
for (each target primitive in the group)
    glStencilMask(0);
    glStencilFunc(GL_EQUAL, 1, Sa);
    glCullFace(GL_BACK);
    <draw the target primitive>
    glStencilMask(Sa);
    glClearStencil(0);
    glClear(GL_STENCIL_BUFFER_BIT);
    a++;
```

Further techniques are available for adding clipping planes (half-spaces), including more normalization rules and pruning opportunities [101]. This is especially important in the case of the near clipping plane in the viewing frustum.

Source code for dynamically loadable Inventor objects implementing this technique is available at the Martin Center at Cambridge web site [102].

4 Geometry and Transformations

OpenGL has a simple and powerful transformation model. Since the transformation machinery in OpenGL is exposed in the form of the modelview and projection matrices, it is possible to develop novel uses for the transformation pipeline. This section describes some useful transformation techniques, and provides some additional insight into the OpenGL graphics pipeline.

4.1 Stereo Viewing

Stereo viewing is a common technique to increase visual realism or enhance user interaction with 3D scenes. Two views of a scene are created, one for the left eye, one for the right. Some sort of viewing hardware is used with the display, so each eye only sees the view created for it. The apparent depth of objects is a function of the difference in their positions from the left and right eye views. When done properly, objects appear to have actual depth, especially with respect to each other. When animating, the left and right back buffers are used, and must be updated each frame.

OpenGL supports stereo viewing, with left and right versions of the front and back buffers. In normal, non-stereo viewing, when not using both buffers, the default buffer is the left one for both front and back buffers. Since OpenGL is window system independent, there are no interfaces in OpenGL for stereo glasses, or other stereo viewing devices. This functionality is part of the OpenGL/Window system interface library; the style of support varies widely.

In order to render a frame in stereo:

- The display must be configured to run in stereo mode.
- The left eye view for each frame must be generated in the left back buffer.
- The right eye view for each frame must be generated in the right back buffer.
- The back buffers must be displayed properly, according to the needs of the stereo viewing hardware.

Computing the left and right eye views is fairly straightforward. The distance separating the two eyes, called the *interocular distance* (*IOD*), must be determined. Choose this value to give the proper spacing of the viewer's eyes relative to the scene being viewed. Whether the scene is microscopic or galaxy-wide is irrelevant. What matters is the size of the imaginary viewer relative to the objects in the scene. This distance should be correlated with the degree of perspective distortion present in the scene to produce a realistic effect.

4.1.1 Fusion Distance

The other parameter is the distance from the eyes where the lines of sight for each eye converge. This distance is called the *fusion distance*. At this distance objects in the scene will appear to be on the front surface of the display ("in the glass"). Objects farther than the fusion distance from the viewer will appear to be "behind the glass" while objects in front will appear to float in front of the display. The latter illusion is harder to maintain, since real objects visible to the viewer beyond the edge of the display tend to destroy the illusion.

Although it is possible to create good looking stereo scenes using dimensionless quantities, the best behavior occurs when everything is measured carefully. This is easy to do if the `glFrustum` call is used rather than the `gluPerspective` call. Pick a unit of measurement, then use those units for screen size, distance from viewer to screen, interocular distance, and so forth. It is a good idea to keep the code that computes the screen parameters separate from the rest of the application, to make it easier to port the program to different screen sizes or arrangements.

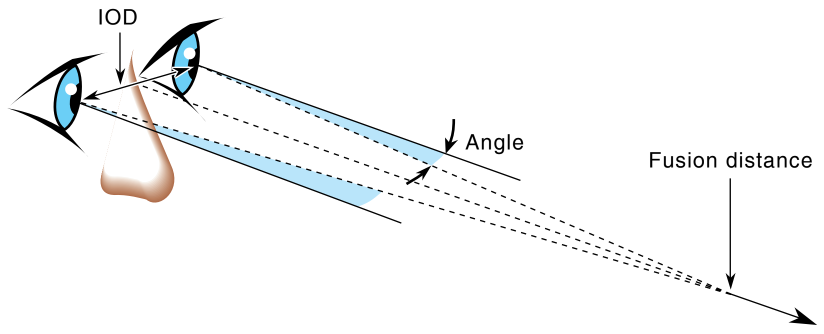


Figure 17. Stereo Viewing Geometry

The view direction vector and the vector separating the left and right eye position are perpendicular to each other. The two view points are located along a line perpendicular to the direction of view and the "up" direction. The fusion distance is measured along the view direction. The position of the viewer can be defined to be at one of the eye points, or halfway between them. In either case, the left and right eye locations are positioned relative to it.

If the viewer is taken to be halfway between the stereo eye positions, and assuming `gluLookAt` has been called to put the viewer position at the origin in eye space, then the fusion distance is measured along the negative z axis (like the near and far clipping planes), and the two viewpoints are on either side of the origin along the x axis, at (-*IOD*/2, 0, 0) and (*IOD*/2, 0, 0).

4.1.2 Computing the Transforms

The transformations needed for correct stereo viewing are simple translations and off-axis projections [22]. Computationally, the stereo viewing transforms happen last, after the viewing transform has been applied to put the viewer at the origin. Since the matrix order is the reverse of the order of operations, the viewing matrices should be applied to the modelview matrix first.

The order of matrix operations should be:

1. Transform from viewer position to left eye view.
2. Apply viewing operation to get to viewer position (`gluLookAt` or equivalent).
3. Apply modeling operations.
4. Change buffers, repeat for right eye.

Assuming that the identity matrix is on the modelview stack and that we want to look at the origin from a distance of `EYE_BACK`:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); /* the default matrix */
glPushMatrix()
glDrawBuffer(GL_BACK_LEFT)
gluLookAt(-IOD/2.0, 0.0, EYE_BACK,
  0.0, 0.0, 0.0,
  0.0, 1.0, 0.0);
```

```
<viewing transforms>
<modeling transforms>
draw()
glPopMatrix();
glPushMatrix()
glDrawBuffer(GL_BACK_RIGHT)
gluLookAt(IOD/2.0, 0.0, EYE_BACK,
  0.0, 0.0, 0.0,
  0.0, 1.0, 0.0);
<viewing transforms>
<modeling transforms>
draw()
glPopMatrix()
```

This method of implementing stereo transforms changes the viewing transform directly using a separate call to `gluLookAt` for each eye view. Move fusion distance along the viewing direction from the viewer position, and use that point for the center of interest of both eyes. Translate the eye position to the appropriate eye, then render the stereo view for the corresponding buffer. This method is quite simple when real-world measurements are used.

An alternative, but less correct, method of implementing stereo transforms is to translate the views left and right by half of the interocular distance, then rotate by the inverse tangent of the ratio between the fusion distance and half of the interocular distance: $angle = \arctan(\frac{fusiondistance}{\frac{IOD}{2}})$ With this method, each viewpoint is rotated towards the centerline halfway between the two viewpoints.

4.2 Depth of Field

Normal viewing transforms act like a perfect pinhole camera; everything visible is in focus, regardless of how close or how far the objects are from the viewer. To increase realism, a scene can be rendered to vary sharpness as a function of viewer distance, more accurately simulating a camera with a finite depth of field.

Depth-of-field and stereo viewing are similar. In both cases, there is more than one viewpoint, with all view directions converging at a fixed distance along the direction of view. When computing depth of field transforms, however, we only use shear instead of rotation, and sample a number of viewpoints, not just two, along an axis perpendicular to the view direction. The resulting images are blended together.

This process creates images where the objects in front of and behind the fusion distance shift position as a function of viewpoint. In the blended image, these objects appear blurry. The closer an object is to the fusion distance, the less it shifts, and the sharper it appears.

The field of view can be expanded by increasing the ratio between the viewpoint shift and fusion distance. This way objects have to be farther from the fusion distance to shift significantly.

For details on rendering scenes featuring a limited field of view see Section 14.3.

4.3 The Z Coordinate and Perspective Projection

The z coordinates are transformed in the same fashion as the x and y coordinates. After transformation, clipping and perspective division, they occupy the range -1.0 through 1.0. The `glDepthRange` mapping specifies a transformation for the z coordinate similar to the viewport transformation used to map x and y to window coordinates. The `glDepthRange` mapping is somewhat different from the viewport mapping in that the hardware resolution of the depth buffer is hidden from the application. The parameters to the `glDepthRange` call are in the range [0.0, 1.0]. The z or depth associated with a fragment represents the distance to the eye. By default the fragments nearest the eye (the ones at the near clip plane) are mapped to 0.0 and the fragments farthest from the eye (those at the far clip plane) are mapped to 1.0. Fragments can be mapped to a subset of the depth buffer range by using smaller

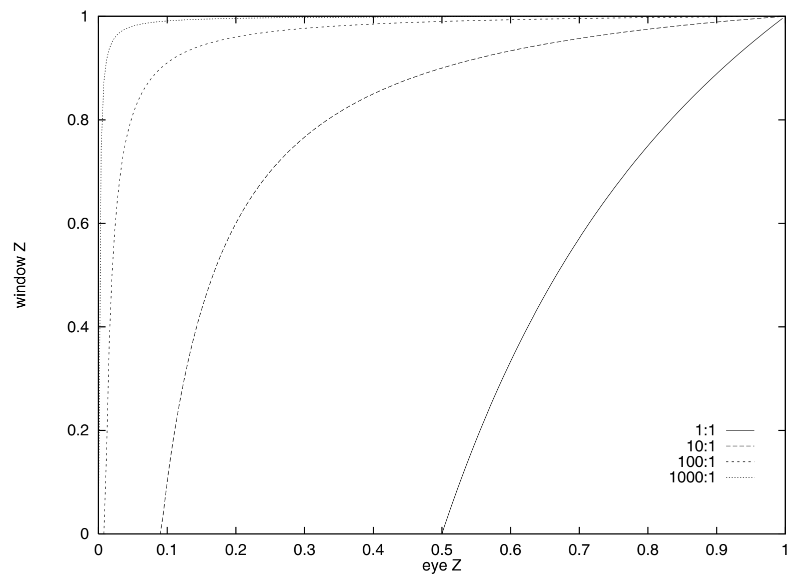


Figure 18. Window z to Eye z Relationship for near/far Ratios

values in the `glDepthRange` call. The mapping may be reversed so that fragments furthest from the eye are at 0.0 and fragments closest to the eye are at 1.0 simply by calling `glDepthRange(1.0,0.0)`. While this reversal is possible, it may not be well-suited for some depth buffer implementations. Parts of the underlying architecture may have been tuned for the forward mapping and may not produce results of the same quality when the mapping is reversed.

To understand why there might be this disparity in the rendering quality, it is important to understand the characteristics of the window z coordinate. The z value specifies the distance from the fragment to the plane of the eye. The relationship between distance and z is linear in an orthographic projection, but not in a perspective projection. In the case of a perspective projection, the amount of the non-linearity is proportional to the ratio of far to near in the `glFrustum` call (or zFar to zNear in the `gluPerspective` call). Figure 18 plots the window coordinate z value as a function of the eye-to-pixel distance for several ratios of far to near. The non-linearity increases the resolution of the z-values when they are close to the near clipping plane, increasing the resolving power of the depth buffer, but decreasing the precision throughout the rest of the viewing frustum, thus decreasing the accuracy of the depth buffer in the back part of the viewing volume.

For objects a given distance from the eye, however, the depth precision is not as bad as it looks in Figure 18. No matter how far back the far clip plane is, at least half of the available depth range is present in the first "unit" of distance. In other words, if the distance from the eye to the near clip plane is one unit, at least half of the z range is used up in the first "unit" from the near clip plane towards the far clip plane. Figure 19 plots the z range for the first unit distance for various ranges. With a million to one ratio, the z value is approximately 0.5 at one unit of distance. As long as the data is mostly drawn close to the near plane, the z precision is good. The far plane could be set to infinity without significantly changing the accuracy of the depth buffer near the viewer.

To achieve the best depth buffer precision, the near plane should be moved as far from the eye as possible without touching the object, which would cause part or all of it to be clipped away. The position of the near clipping plane has no effect on the projection of the x and y coordinates and therefore has minimal effect on the image.

Putting the near clip plane closer to the eye than to the object results in loss of depth buffer precision.

In addition to depth buffering, the z coordinate is also used for fog computations. Some implementations may

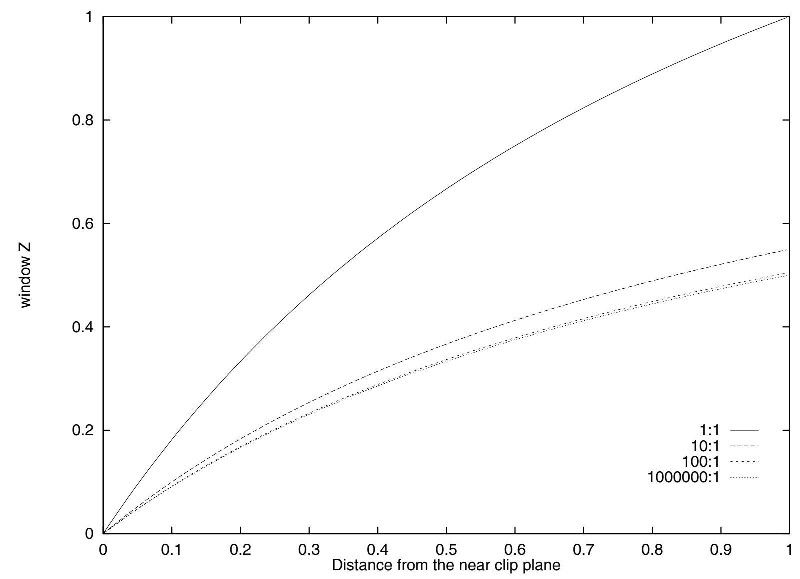


Figure 19. Available Window z Depth Values near/far Ratios

perform the fog computation on a per-vertex basis using eye z and then interpolate the resulting colors whereas other implementations may perform the computation for each fragment. In this case, the implementation may use the window z to perform the fog computation. Implementations may also choose to convert the computation into a cheaper table lookup operation which can also cause difficulties with the non-linear nature of window z under perspective projections. If the implementation uses a linearly indexed table, large far to near ratios will leave few table entries for the large eye z values. This can cause noticeable Mach bands in fogged scenes.

4.3.1 Depth Buffering

We have discussed some of the caveats of using depth buffering, but there are several other aspects of OpenGL rasterization and depth buffering that are worth mentioning [2]. One big problem is that the rasterization process uses inexact arithmetic so it is exceedingly difficult to handle primitives that are coplanar unless they share the same plane equation. This problem is exacerbated by the finite precision of depth buffer implementations. Many solutions have been proposed to handle this class of problems, which involve coplanar primitives:

1. Decaling
2. Hidden line elimination
3. Outlined polygons
4. Shadows

Many of these problems have elegant solutions involving the stencil buffer (Section 7.2, Section 3.5), but it is still worth describing alternative methods to get more insight into the uses of the depth buffer.

The problem of decaling one coplanar polygon into another can be solved rather simply by using the painter's algorithm (i.e., drawing from back to front) combined with color buffer and depth buffer masking, assuming the decal is contained entirely within the underlying polygon. The steps are:

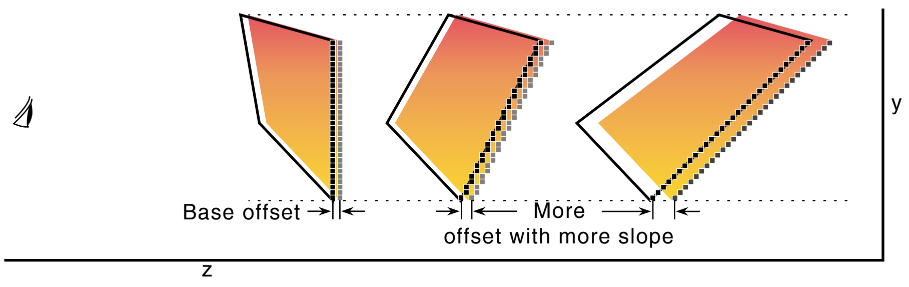


Figure 20. Polygon and Outline Slopes

1. Draw the underlying polygon with depth testing enabled but depth buffer updates disabled.
2. Draw the top layer polygon (decal) also with depth testing enabled and depth buffer updates still disabled.
3. Draw the underlying polygon one more time with depth testing and depth buffer updates enabled, but color buffer updates disabled.
4. Enable color buffer updates and continue on.

Outlining a polygon and drawing hidden lines are similar problems. If we have an algorithm to outline polygons, hidden lines can be removed by outlining polygons with one color and drawing the filled polygons with the background color. Ideally a polygon could be outlined by simply connecting the vertices together with line primitives. This seems similar to the decaling problem except that edges of the polygon being outlined may be shared with other polygons and those polygons may not be coplanar with the outlined polygon, so the decaling algorithm can not be used, since it relies on the coplanar decal being fully contained within the base polygon.

The solution most frequently suggested for this problem is to draw the outline as a series of lines and translate the outline a small amount towards the eye. Alternately, the polygon could be translated away from the eye instead. Besides not being a particularly elegant solution, there is a problem in determining the amount to translate the polygon (or outline). In fact, in the general case there is no constant amount that can be expressed as a simple translation of the z object coordinate that will work for all polygons in a scene.

Figure 20 shows two polygons (solid) with outlines (dashed) in the screen space y-z plane. One of the primitive pairs has a 45-degree slope in the y-z plane and the other has a very steep slope. During the rasterization process the depth value for a given fragment may be derived from a sample point nearly an entire pixel away from the edge of the polygon. Therefore the translation must be as large as the maximum absolute change in depth for any single pixel step on the face of the polygon. The figure shows that the steeper the depth slope, the larger the required translation. If an unduly large constant value is used to deal with steep depth slopes, then for polygons which have a shallower slope there is an increased likelihood that another neighboring polygon might end up interposed between the outline and the polygon. So it seems that a translation proportional to the depth slope is necessary. However, a translation proportional to slope is not sufficient for a polygon that has constant depth (zero slope) since it would not be translated at all. Therefore a bias is also needed. Many vendors have implemented the `EXT_polygon_offset` extension that provides a scaled slope plus bias capability for solving outline problems such as these and for other applications. A modified version of this polygon offset extension has been added to the core of OpenGL 1.1 as well.

4.4 Image Tiling

When rendering a scene in OpenGL, the resolution of the image is normally limited to the workstation screen size. For interactive applications this is usually sufficient, but there may be times when a higher resolution image is needed. Examples include color printing applications and computer graphics recorded for film. In these cases, higher resolution images can be divided into tiles that fit on the workstation's framebuffer. The image is rendered tile by tile, with the results saved into off screen memory, or perhaps a file. The image can then be sent to a printer or film recorder, or undergo further processing, such has downsampling to produce an antialiased image.

One straightforward way to tile an image is to manipulate the `glFrustum` call's arguments. The scene can be rendered repeatedly, one tile at a time, by changing the left, right, bottom and top arguments arguments of `glFrustum` for each tile.

Computing the argument values is straightforward. Divide the original width and height range by the number of tiles horizontally and vertically, and use those values to parametrically find the left, right, top, and bottom values for each tile.

$$\begin{aligned}
tile(i,j); i: 0 \rightarrow nTiles_{horiz}, j: 0 \rightarrow nTiles_{vert} \\
right_{tiled}(i) = left_{orig} + \frac{right_{orig} - left_{orig}}{nTiles_{horiz}} * (i+1) \\
left_{tiled}(i) = left_{orig} + \frac{right_{orig} - left_{orig}}{nTiles_{horiz}} * i \\
top_{tiled}(j) = bottom_{orig} + \frac{top_{orig} - bottom_{orig}}{nTiles_{vert}} * (j+1) \\
bottom_{tiled}(j) = bottom_{orig} + \frac{top_{orig} - bottom_{orig}}{nTiles_{vert}} * j
\end{aligned}$$

In the equations above, each value of i and j corresponds to a tile in the scene. If the original scene is divided into $nTiles_{horiz}$ by $nTiles_{vert}$ tiles, then iterating through the combinations of i and j generate the left, right top, and bottom values for `glFrustum` to create the tile.

Since `glFrustum` has a shearing component in the matrix, the tiles stitch together seamlessly to form the scene. Unfortunately, this technique would have to be modified for use with `gluPerspective` or `glOrtho`. There is a better approach, however. Instead of modifying the perspective transform call directly, apply transforms to the results. The area of normalized device coordinate (NDC) space corresponding to the tile of interest is translated and scaled so it fills the NDC cube. Working in NDC space instead of eye space makes finding the tiling transforms easier, and is independent of the type of projective transform.

Even though it is easy to visualize the operations happening in NDC space, conceptually, you can "push" the transforms back into eye space, and the technique maps into the `glFrustum` approach described above.

For the transform operations to happen after the projection transform, the OpenGL calls must happen before it. Here is the sequence of operations:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glScalef(xScale, yScale);
glTranslatef(xOffset, yOffset, 0.f);
setProjection();
```

The scale factors xScale and yScale scale the tile of interest to fill the the entire scene:

$$xScale = \frac{sceneWidth}{tileWidth}$$

$$yScale = \frac{sceneHeight}{tileHeight}$$

The offsets `xOffset` and `yOffset` are used to offset the tile so it is centered about the z axis. In this example, the tiles are specified by their lower left corner relative to their position in the scene, but the translation needs to move the center of the tile into the origin of the x-y plane in NDC space:

$$xOffset = \frac{-2 * left}{sceneWidth} \quad + \quad (1 - \frac{1}{nTiles_{horiz}})$$
$$yOffset = \frac{-2 * bottom}{sceneHeight} \quad + \quad (1 - \frac{1}{nTiles_{vert}})$$

As before $nTiles_{horiz}$ is the number of tiles that span the scene horizontally, while $nTiles_{horiz}$ is the number of tiles that span the scene vertically.

Some care should be taken when computing $left$, $bottom$, $tileWidth$ and $tileHeight$ values. It is important that each tile is abutted properly with its neighbors. Ensure this by guarding against round-off errors. Some code that properly computes these values is given below:

```
/* tileWidth and tileHeight are GLfloats */
GLint bottom, top;
GLint left, right;
GLint width, height;
for(j = 0; j < num_vertical_tiles; j++) {
        for(i = 0; i < num_horizontal_tiles; i++) {
                left = i * tileWidth;
                right = (i + 1) * tileWidth;
                bottom = j * tileHeight;
                top = (j + 1) * tileHeight;
                width = right - left;
                height = top - bottom;
                /* compute xScale, yScale, xOffset, yOffset */
        }
}
```

Note that the parameter values are computed so that $left + tileWidth$ is guaranteed to be equal to $right$ and equal to $left$ of the next tile over, even if $tileWidth$ has a fractional component. If the frustum technique is used, similar precautions should be taken with the $left$, $right$, $bottom$, and top parameters to `glFrustum`.

4.5 Moving the Current Raster Position

Using the `glRasterPos` command, the raster position will be invalid if the specified position was culled. Since `glDrawPixels` and `glCopyPixels` operations applied when the raster position is invalid do not draw anything, it may seem that the lower left corner of a pixel rectangle must be inside the clip rectangle. This problem may be overcome by using the `glBitmap` command. The `glBitmap` command takes arguments `xoff` and `yoff` which specify an increment to be added to the current raster position. Assuming the raster position is valid, it may be moved outside the clipping rectangle by a `glBitmap` command. `glBitmap` is often used with a zero size rectangle to move the raster position.

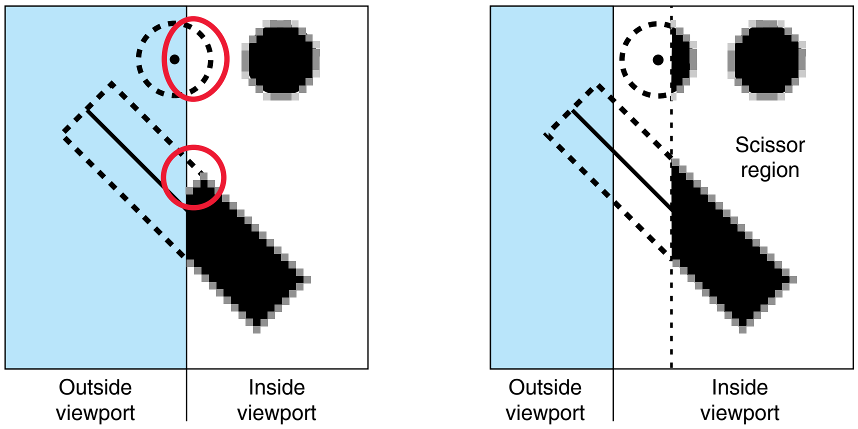


Figure 21. Clipped Wide Primitives Can Still be Visible

4.6 Preventing Clipping of Wide Lines and Points

It is important to note that OpenGL points are clipped if their projected position is beyond the viewport. If a point size other than 1 is specified with `glPointSize`, the object will appear to "pop" out of view when the center of the wide point exits the viewport. This is because the point itself has no area, and as such is clipped based solely on its position. An example scenario is shown in Figure 21.

Wide lines have the same problem. The line is clipped to the viewport, and thus some pixels contributed by the original line are no longer drawn, as shown in Figure 21.

This problem is more significant in a multiple-display setting, such as a three-monitor flight simulator, or in a multiple-viewport setting such as a cylindrical projection.

These missing pixels can be restored by setting the scissor region to the visible area and then enlarging the viewport so that points and lines are clipped beyond the region in which they could contribute pixels. For n-pixel wide points and lines, this margin is $n - 1$ pixels. The viewing frustum has to be enlarged based on the new viewport so that points are rasterized to the same pixels within the larger viewport and scissor region as they were in the smaller viewport.

4.7 Distortion Correction

A workstation user with a single monitor and a monoptic visual will usually sit in a location relative to his or her screen that closely approximates the single symmetric frustum typically supplied to OpenGL as the view model.

In visual simulation applications with curved screens ("domes"), virtual reality "caves" and the like, and any situation where the projection unit, projection surface, and viewing parameters don't correspond to a symmetric static frustum, some correction will be required to make the visible image seem accurate and visibly consistent.

Visual inaccuracy is caused by the difference between the observer's view of the surface and the video projector's view of the surface, and is exacerbated by a non-planar screen surface, such as a spherical shell.

If the display surface has no skew component to it, like an ordinary computer monitor or a video projector which is aligned perpendicular to the screen, but the observer's view direction is not perpendicular to the screen, use an asymmetric frustum. This can be accomplished by providing appropriate $left$, $right$, top, and $bottom$ parameters to `glFrustum` that form a near plane which is not centered on the z axis.

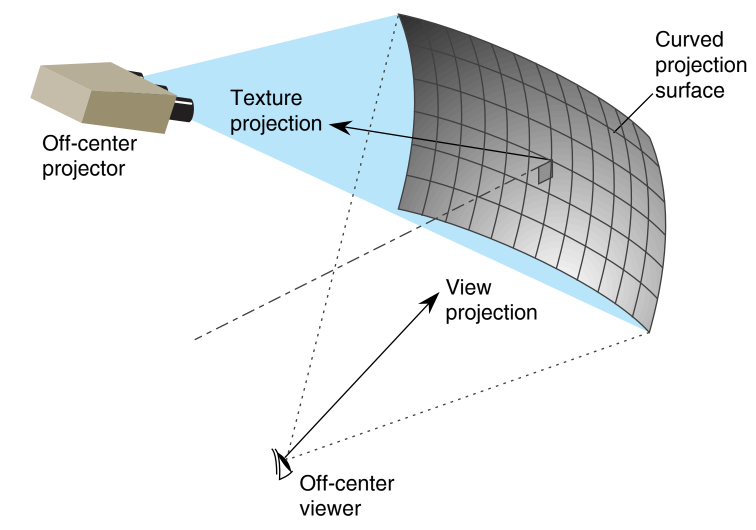


Figure 22. A Complex Display Configuration

If the display surface is askew, as it is if the projector is located above the observer as in a movie theatre, the perspective distortion in the projection must be corrected. This can be accomplished by rendering the scene using an asymmetric frustum as above, storing the rendered scene as a texture, and then drawing a quad textured scene with a projective texture matrix corresponding to the off-center video projector frustum.

Finally, if the display surface itself is non-planar, like the spherical and cylindrical screens used in some flight simulators, a combination of the above technique and image warping is required to produce an accurate image.

- Create a uniform grid as viewed by the observer.
- Project the vertices of the grid onto the screen surface.
- Project the vertices from the screen surface onto a plane perpendicular to the display direction of the video projector.
- Store the projected vertices' normalized viewing coordinates $[0, 1)$ on that plane as texture coordinates for the original grid.
- Render the scene normally from the viewpoint of the observer.
- Transfer the image into a texture.
- Render the image textured onto the uniform grid with the warped texture vertices.

You may have to render a larger image than will finally be viewed so that the warped image does not contain any blank areas.

For further information on imagewarping and dewarping, see Section 6.18.

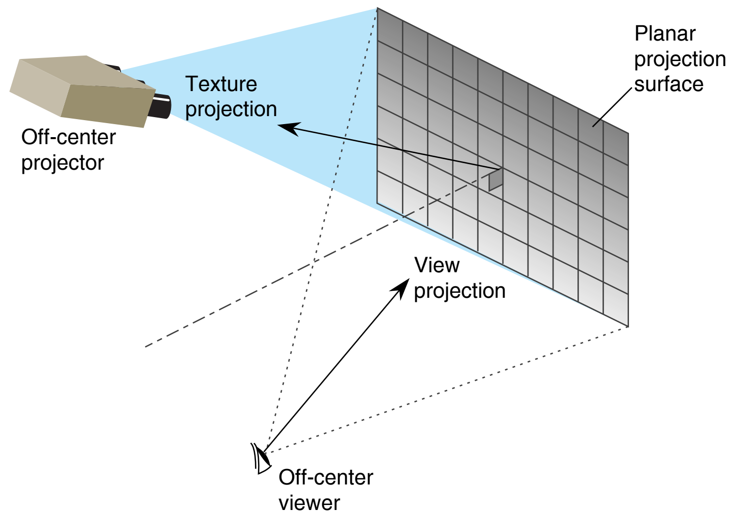


Figure 23. A Configuration with Off-Center Projector and Viewer

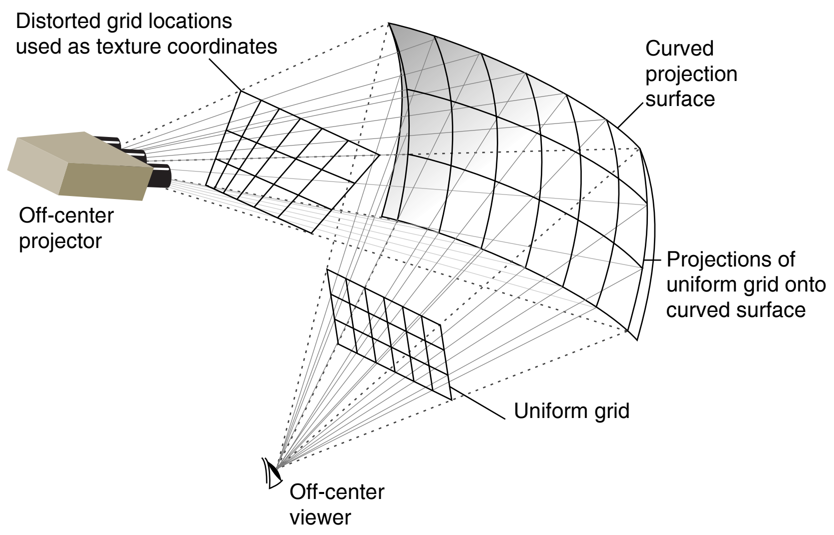


Figure 24. Distortion Correction Using Texture Mapping

4.8 Picking and Highlighting

Interactive selection of objects with feedback is an important part of modeling applications. OpenGL provides several mechanisms which can be used to do the perform the object selection and the highlighting tasks.

4.8.1 OpenGL Selection

OpenGL supports an object selection mechanism in which the object geometry is transformed and compared against a selection subregion of a window. The mechanism uses the transformation pipeline to compare object vertices against the view volume. To reduce the view volume to a screen-space subregion (window coordinates), the projected coordinates of the object are transformed by the following matrix

$$T = \begin{pmatrix} \frac{p_x}{d_x} & 0 & 0 & p_x - 2\frac{q_x - o_x}{d_x} \\ 0 & \frac{p_y}{d_y} & 0 & p_y - 2\frac{q_y - o_y}{d_y} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

where o_x, o_y, p_x, and p_y are the x and y origin and width and height of the viewport, and q_x, q_y, d_x, and d_y are the origin and width and height of the pick region.

Objects are identified by assigning them integer names using `glLoadName`. As each object is sent to the OpenGL pipeline and tested against the pick region, if the test succeeds a *hit record* is created to identify the object. The hit record is written to the selection buffer whenever a change is made to the current object name. An application can determine which objects intersected the pick region by scanning the selection buffer and examining the names present in the buffer.

The OpenGL selection method determines that an object has been hit if it intersects the view volume. Bitmap and pixel image primitives will generate a hit record only if a raster positioning command is sent to the pipeline and the transformed position lies within the viewing volume. To generate hit records for any point within a pixel image or bitmap, a bounding rectangle should be sent rather than the image, and selection testing performed on the interior of the rectangle. Similarly, wide lines and points will be selected only if the equivalent infinitely thin line or small point would be selected. To facilitate selection testing of wide lines and points, proxy geometry representing the true footprint of the primitive should be used instead.

Many applications take advantage of instancing of geometric data to reduce the memory footprint. Instancing allows an application to create a single representation of the geometric data for each type of object used in the scene. If the application is modeling a car, the four wheels of the car may be represented as instances of a single geometric description of a wheel combined with a modeling transformation to place the wheel in the correct location in the scene. Instancing introduces extra complexity into the picking operation. If a single name is associated with the wheel geometry, the application can not determine which of the four instances of the wheel has been picked. OpenGL solves this problem by maintaining a stack of object names. This allows an application that represents a model hierarchically to associate a name at each stage in the hierarchy. As the hierarchical model is drawn new names are pushed onto the stack as the hierarchy is descended and old names are popped as the hierarchy is ascended. When a hit record is created, it contains all of the names currently in the name stack. The application can determine which instance of an object is selected by looking at the contents of the name stack and comparing them to the names stored in the hierarchical representation of the model.

Using the car model example, the application associates an object name with the wheel representation and another object name with each of the transformations used to position the wheel in the car model. The application determines that a wheel is selected if the selection buffer contains the object name for the wheel, and it determines which instance of the wheel by examining the object name of the transformation.

When the OpenGL pipeline is in selection mode, the primitives sent to the pipeline do not generate fragments to the framebuffer. Since only the result of the transformation pipeline is of interest, there is no need to send texture coordinates, normals, etc. or to enable lighting.

4.8.2 Object Tagging in the Color Buffer

An alternative method for locating objects is to write integer object names as color values into the framebuffer and read back the framebuffer data within the pick region to reconstruct the object names. In order for this to work correctly, the application needs to be able to predictably write and read color values. Texturing, blending, dithering, lighting and smooth shading should be disabled so that fragment color values are not altered during rasterization or fragment processing. The unsigned integer forms of the color commands (`glColor3ub`, etc) should be used to pass in the object names, as the unsigned forms are specified to convert the values in such a way as to preserve the *b* most significant bits of the color value, where *b* is the number of bits in the color buffer. To limit selection to visible surfaces, depth testing should be enabled. The back color buffer can be used for the drawing operations to keep the drawing operations invisible to the user.

A typical RGB color buffer storing 8-bit components can represent 24-bit object names. However, since the color must uniquely identify the object, instancing information must be available directly in the color. That is, the functionality provided by the name stack in the OpenGL selection mechanism is not available, so the application may need to partition the representable name space to hold hierarchy information. For example a 4 level hierarchy can be allocated a 24-bit color as 4,4,6 and 10 bits allowing more object names for leaf parts of the hierarchy.

One disadvantage of using the color buffer is that the color buffer can only hold a single identifier at each pixel. If depth buffering is used, then the pixel will hold the object name corresponding to a visible surface. If depth buffering is not used, then a pixel hold the name of the last surface drawn. The OpenGL selection mechanism can return a hit record for all objects that intersect a given region. The application is free to choose one of the intersecting objects using a separate policy, e.g. the object closest to the viewer, iterate through all of the objects one at a time, etc.

4.8.3 Proxy Geometry

One method to reduce the amount of work done by the OpenGL pipeline during picking operations (for color buffer tagging or OpenGL selection) is to use a simplified form of the object in the picking computations. For example, individual objects can be replaced by geometry representing their bounding boxes. The accuracy of the picking operation is traded for increased speed. Some of the accuracy can be improved by adding a second pass in which the objects which are selected using their simplified geometry are reprocessed using their real geometry.

4.8.4 Other Methods

For many applications it may prove advantageous to not use the OpenGL pipeline at all to implement picking. For example, an application may choose to organize its geometric data spatially and use a hierarchy of bounding volumes to efficiently prune portions of the scene without testing each individual object [83, 93].

4.8.5 Highlighting

Once the selected object has been identified, an application will typically modify the appearance of the object to indicate that it has been selected. A simple way to accomplish this is to redraw the entire scene, drawing the selected object with a different appearance (wireframe, different color, etc).

In applications manipulating complex models, the cost of redrawing the entire scene to indicate a selection may be prohibitive. This is particularly true for applications which implement *locate-highlight* in which each object is highlighted as the cursor passes over or near it to indicate that this is the candidate object for manipulation by the application. An extension of this problem exists for painting applications that need to track the location of a brush over an object and make changes to the appearance of the object based on the current painting parameters [46].

An alternative to redrawing the entire scene is to use overlay color buffers to draw highlights on top of the existing scene. One difficulty with this strategy is that it may be difficult to modify only the visible surfaces of the selected object since the depth information is present in the depth buffer associated with the main color buffer. For applications in which the visible surface information is not required, overlay buffers are an efficient solution. If visible surface information is important, then it may be better to modify the color buffer directly. A selected object that has been depth buffered can be overdrawn directly by changing the depth test function to `GL_LEQUAL` and redrawing the object geometry with different attributes.

4.8.6 XOR Highlighting

Another efficient highlighting technique is to overdraw with an XOR logic operation. An advantage of using XOR is that the highlighting and restoration operations can be done independently of the original object color. The most significant bit of each of the color components can be XORed to produce a large difference between the highlight color and the original color. Drawing a second time restores the original color.

A second advantage of the XOR method is that depth testing can be disabled to allow the non-visible surfaces to poke through occluding objects. The highlight can be later removed without needing to redraw the occluders.

One should also be careful of interactions between the picking and highlighting methods. For example, a picking mechanism that uses the color or depth buffer can not be mixed with a highlighting algorithm that relies on the contents of those buffers remaining intact between highlighting operations.

A useful hybrid scheme for combining color buffer tagging with locate-highlight on visible surfaces is to share the depth buffer between the picking and highlighting operations and to use the front color buffer for highlighting operations and the back color buffer for locate operations. Each time the viewing or modeling transformations change, the scene is redrawn updating both color buffers and locate-highlight operations are performed using the same buffers until another modeling or viewing change requires a redraw. This type of algorithm can be very effective for achieving interactive rates for complex models since very little geometry needs to be rendered between modeling and viewing changes.

4.9 Foreground Object Manipulation

The schemes for fast redrawless highlighting can be generalized to allow a limited form of manipulation of a selected depth buffered object (a *foreground* object) while avoiding full scene redraw. The main idea is that the entire scene except for the foreground object is drawn updating the color and depth buffers and copies of the depth and color buffers are made. Each time the foreground object is moved or modified, the back buffer and depth buffer are initialized using the saved copies and the foreground object is drawn as normal and is depth buffered .

This image-based technique is similar to the algorithm described for compositing images with depth in Section 8.7. In order for it to work it requires a method to efficiently save and restore the color and depth images for an intermediate form of the scene. If aux buffers or stereo color buffers are available these can be used to store the color buffer (using `glCopyPixels`) and the depth buffer can be saved to the host, or if present, to a pixel buffer (PBuffer). Pbuffers are described in Section 13.1.5. It is particularly important that the contents of the depth buffer be saved and restored accurately. If some of the depth buffer values are truncated or rounded during the transfer, then the resulting image will not be the same as that produced by drawing the entire scene.

This technique works best when the geometric complexity of the scene is very large – so large that the time spent transferring the color and depth buffers is small compared to the amount of time that would be necessary to render the entire scene.

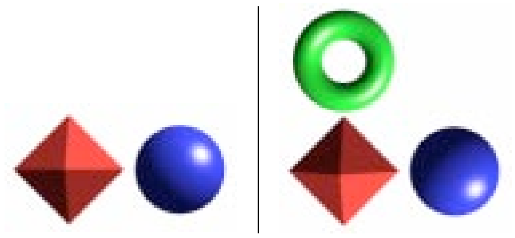


Figure 25. Occluded Torus: Front and Top Views

5 Occlusion Culling

Complex models with high depth complexity render many pixels which are ultimately discarded during depth testing. Transforming vertices and rasterizing primitives which are occluded by other polygons reduces the frame rate of the application while adding nothing to the visual quality of the image. Occlusion culling algorithms attempt to identify such non-visible polygons and discard them before they are sent to the rendering pipeline [19, 106]. Occlusion culling algorithms are a form of visible surface determination algorithms that attempt to resolve visible (or non-visible surfaces) at larger granularity than pixel-by-pixel testing.

A simple example of an occlusion culling algorithm is backface culling. The surfaces of a closed object which are pointing away from the viewer will be occluded by the surfaces pointing towards the viewer, so there is no need to draw them. Many occlusion culling algorithms operate in object space [19, 60] and there is little that can be done with the standard OpenGL pipeline to accelerate such operations. However, Zhang et. al. [105] describe an algorithm which computes a hierarchy of image-space *occlusion maps* to use in testing whether polygons comprising the scene are visible.

An occlusion map is a 2D array of values, where each value measures the opacity of the image plane at that point. An occlusion map corresponding to a set of geometry is generated by rendering the geometry with the polygon faces colored white. The occlusion map is view dependent. In Zhang's algorithm the occlusion map is generated from a target set of occluders. The occlusion map is accompanied by a *depth estimation buffer* that provides a conservative estimate of the maximum depth value of a set of occluders at each pixel. Together the occlusion map and depth estimation buffer are used to determine whether a candidate object is occluded. A bounding volume for the candidate object is projected onto the same image plane as the occlusion map and the resulting projection is compared against the occlusion map to determine whether the occluders overlap the portion of the image where the object would be rendered. If the object is determined to be overlapped by the occluders, the depth estimation buffer is tested to determine whether the candidate object is behind the occluder geometry. A pyramidal hierarchy of occlusion maps (similar to a mipmap hierarchy) can be constructed to accelerate the initial overlap tests.

5.1 Choosing Occluders

Choosing a good set of occluders can be computationally expensive as it is approximating the task of determining the visible surfaces. Heuristic methods can be used to choose likely occluders based on an estimation of the size of the occluder and distance from they eye. In order to maintain interactive rendering it may be useful to assign a fixed polygon budget to the list of occluders. Temporal coherence can be exploited to reduce the number of new occluders that need to be considered each frame.

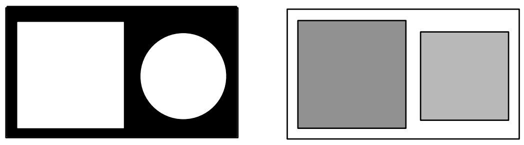


Figure 26. Occlusion Map and Depth Estimation Buffer

5.2 Building the Occlusion Map

Once the occluders have been selected they are rendered to the framebuffer with lighting and texturing disabled. The polygons are colored white to produce values near or equal to 1.0 in opaque areas. OpenGL implementations which do not support some form of antialiasing will have pixels values that are either 0.0 or 1.0. A hierarchy of reduced resolution maps is created by copying this map to texture memory and performing bilinear texture filtering on the image to produce an image 1/4 the size. Additional maps are created by repeating this process. The size of the highest resolution map and the number of hierarchy levels created is a compromise between the amount of time spent rendering, copying, and reading back the images and the accuracy of the result. Some of the lower resolution images may be more efficiently computed on the host processor as the amount of overhead involved in performing copies to framebuffer or pixel readback operation dominates the time spent producing the pixels.

5.3 Building the Depth Estimation Buffer

In [105], Zhang suggests building a depth estimation buffer by computing the farthest depth value in the projected bounding box for an occluder and using this value throughout the screen-space bounding rectangle for the occluder. The end result is a tiling of the image plane with a set of projected occluders each representing a single depth value, as shown in Figure 26. The computation is kept simple to avoid complex scan-conversion of the occluder and to simplify the depth comparisons against a candidate occluded object.

5.4 Occlusion Testing

The algorithm for occlusion testing consists of two steps. First the screen space bounding rectangle of the potentially occluded object is computed and tested for overlap against the hierarchy of occlusion maps. If the occluders overlap the object, then a conservative depth value (minimum depth value) is computed for the screen space bounding rectangle of the candidate object. This depth value is tested against the the depth estimation buffer to determine whether the candidate is behind the occluders and is therefore occluded.

Each opacity value in a level in the occlusion map hierarchy corresponds to the coverage of the corresponding screen region. In the general case, the opacity values will range between 0.0 and 1.0 and values between the extrema correspond to screen regions which are partially covered by the occluders. To determine whether a candidate object is occluded, the overlap test is performed against a map level using the candidate's bounding rectangle. If the region corresponding to the candidate is completely opaque in the occlusion map, then the candidate is occluded if it lies behind the occluders (using the depth estimation buffer). The occlusion map hierarchy can be used to accelerate the testing process by starting at the low resolution maps and progressing to higher resolution maps when there is ambiguity.

Since the opacity values provide an estimation of coverage, they can also be used to do more aggressive occlusion culling by pruning candidate objects which are not completely occluded using a threshold opacity value. Since opacity values are generated using simple averaging, the threshold value can be correlated to a bound on the largest *hole* in the occluder set. However, the opacity value is a measure of the number of non-opaque pixels and provides no information on the distribution of those pixels. Aggressive culling is advantageous for scenes with a large number of occluders that do not completely cover the candidates, e.g. a wall of trees. However, if there is a large color discontinuity between the culled objects and the background, distracting popping artifacts may result as the view is changed and aggressively culled objects appear and reappear.

5.5 Other Methods

Zhang's algorithm maintains the depth estimation buffer using simplified software scan conversion and uses the OpenGL pipeline to optimize the computation of the occlusion maps. All testing is performed on the the host, which has the advantage that the testing can be performed asynchronously with the drawing operations and they test results can be computed with very low latency. Another possibility is to maintain the occlusion buffer in the hardware accelerator itself. In order to be useful, there must be a method for testing the screen-space bounding rectangle against the map and efficiently return the result to the application.

The OpenGL depth buffer can be used to do this with some additional extensions. Occluders are selected using the heuristics described above and rendered to the framebuffer as regular geometry. Following this, bounding geometry for candidate objects are rendered and tested against the depth buffer without changing the contents of the color buffer or depth buffer. The result of the depth test is then returned to the application, preferably reduced to a single value rather than the results of the depth test for every fragment generated. The results of the tests are used to determine whether to draw the candidate geometry or discard it. Extensions for performing the occlusion test and returning the result have been proposed and implemented by several hardware vendors [74, 13] and more implementations are likely in the future.

6 Texture Mapping

Texture mapping is one of the primary techniques to improve the appearance of objects rendered with OpenGL. Texturing is typically used to provide color detail for intricate surfaces by modifying the surface color. For example, a woodgrain supplied by a texture can make a flat polygon appear to be made of wood. Current 3D video games now use texture mapping extensively. Texturing can also be the basis for many more sophisticated rendering algorithms for improving visual realism and quality. For example, environment mapping is a view-dependent texture mapping technique that supplies a specular reflection to the surface of objects. This makes it appear that the environment is reflected in the object. More generally texturing can be thought of as a method of providing (or perturbing) parameters to the shading equation such as the surface normal (bump mapping), or even the coordinates of the point being shaded (displacement mapping) based on a parameterization of the surface defined by the texture coordinates. OpenGL readily supports the first two techniques (surface color manipulation and environment mapping). Texture mapping, using bump mapping, can also solve some rendering problems in less obvious ways. This section reviews some of the details of OpenGL texturing support, outlines some considerations when using texturing and suggests some interesting algorithms using texturing.

6.1 Texturing Basics

6.1.1 The Texture Image

The meat of a texture is the texture's image. This is a array of color values. The color values of a texture are referred to as *texels* (short for texture elements and a pun on the word pixel). The texture image array is typically 1D or 2D, however OpenGL 1.2 adds support for 3D texture images as well.[1] The OpenGL `SGIS_texture4D` extension even provides the option for 4D texture images.

The `glTexImage1D`, `glTexImage2D`, and `glTexImage3D` commands specify a complete texture image. The commands copy the texture image data from the application's address space into texture memory. OpenGL's pixel store unpack state determines how the texture image is arranged in memory. Other OpenGL commands update rectangular subregions of an existing texture image (subtexture loads). Still other texture commands copy color data from the frame buffer into texture memory.

Typically, texture images are loaded from image files stored using a standard 2D image file format such as TIFF or JPEG. To make an image file into a texture for use by OpenGL, the OpenGL application is responsible for reading and decompressing as necessary the image file. Once the image is in memory as an uncompressed array, `glTexImage2D` can be passed the size, format, and pointer to the image in memory. The OpenGL API limits itself to rendering functionality and therefore has no support for loading image files. You can either write an image loader yourself or use one of the numerous image loading libraries that are widely available. In addition to loading image files, applications are free to compute or otherwise procedurally generate texture images. Some techniques for procedural texture generation are discussed in Section 6.20.2. Rendering the image using OpenGL and then copying the image from the framebuffer with `glCopyTexImage2D` is yet another option.

OpenGL's pixel transfer pipeline can process the texture image data when texture images are specified. While typically the pixel transfer pipeline is configured to pass texture image data through unchanged, operations such as color space conversions can be performed during texture image download. When optimized by your OpenGL implementation, the pixel transfer operations can significantly accelerate various common processing operations applied to texture image data. The pixel transfer pipeline is further described in Sections 13.1.1 and 13.1.4.

[1]The phrase *3D texturing* is often used in touting new graphics hardware and software products. The common usage of the phrase is to indicate support for applying a 2D texture to 3D geometry. OpenGL's specification would call that merely *2D texturing*. OpenGL assumes that any type of texturing can be applied to arbitrary 3D geometry so the dimensionality of texture mapping (1D, 2D, or 3D) is based on the dimensionality of the texture image. A 2D texture image (one with width and height) is used for 2D texturing. A 3D image (one with width, height, and depth) is required for 3D texturing in the OpenGL technical sense of the phrase. Unfortunately, the market continues to use the phrase *3D texturing* to mean just `GL_TEXTURE_2D`. To avoid confusion, the phrase *volumetric texturing* unambiguously refers to what OpenGL technically calls 3D texturing. Be aware that the phrases *solid texture* and *hypertexture* are also used in the graphics literature to denote 3D texture images. One more bit of trivia: the term *voxel* is often used to denote the texels of a 3D texture image.

The width, height, and depth of a texture image without a border must be powers of two. A texture with a border has an additional one pixel border around the edge of the texture image proper. Since the border is on each side, the border adds two texels in each texture dimension. The rationale for texture images with borders will be discussed in Section 6.4. The texels that make up the texture image have a particular color format. The color format options are RGB, RGBA, luminance, intensity, and luminance-alpha. Sized versions of the texture color formats permit applications a means to hint to the OpenGL implementation for trading off texture memory requirements with texture color quality.

Internal Texture Formats If you care about the quality of your textures or want to conserve the amount of texture memory your application requires (and often conserving texture memory helps improve performance), you should definitely use appropriate internal formats. Internal texture formats were introduced in OpenGL 1.1. Table 1 lists the available internal texture formats. If your texture is known to be only gray-scale or luminance values, choosing the `GL_LUMINANCE` format instead of `GL_RGB` typically cuts your texture memory usage by one third. Requesting more efficient internal format sizes can also help. The `GL_RGB8` internal texture format requests 8 bits of red, green, and blue precision for each texel. The more space efficient `GL_RGB4` internal texture format uses only 4 bits per component making it require only half the texture memory of the `GL_RGB8` format. Of course, the `GL_RGB4` format only has 16 distinct values per component instead of 256 values for the `GL_RGB8` format. However, if minimizing texture memory usage (and often improving texturing performance too) is more important than better texture quality, the `GL_RGB4` format is a better choice. In the case where the source image for your texture only has 4 bits of color resolution per component, there is absolutely no reason to request a format with more than 4 bits of color resolution.

Some words of advice about internal texture formats: If you do not request a specific internal resolution for your texture image because you requested a `GL_RGBA` internal format instead of a size-specific internal format such as `GL_RGBA8` or `GL_RGBA4`, your OpenGL implementation is free to pick the "most appropriate" format for the particular implementation. If a smaller texture format has better texturing performance, the implementation is free to choose the smaller format. This means if you care about maintaining a particular level of internal format resolution, selecting a size-specific texture format is strongly recommended.

Some words of warning about internal texture formats: Not all OpenGL implementations are expected to support all the available internal texture formats. This means just because you request a `GL_LUMIANCE12_ALPHA4` format (to pick a format that is likely to be obscure) does not mean that your texture is guaranteed to be stored in this format. The size-specific internal texture formats are merely hints. If the best the OpenGL implementation can provide is `GL_LUMINANCE8_ALPHA8`, this will be the format you get, even though is provides less luminance precision and more alpha precision than you requested.

6.1.2 Texture Coordinates

Texture coordinates are the means by which texture image positions are assigned to vertices. The per-vertex assignment of texture coordinates is the key to mapping a texture image to rendered geometry. During rasterization, the texture coordinates of a primitive's vertices are interpolated across the primitive so that each rasterized fragment making up the primitive has an appropriately interpolated texture coordinate. A fragment's texture coordinates are translated into the addresses of one or more texels within the current texture. The texels are fetched and their color values are then filtered into a single texture color value for the fragment. The fragment's texture color is then combined with the fragments color.

The vertices of all primitives (including the raster position of pixel images) have associated texture coordinates. Figure 27 shows how object coordinates have associated texture coordinates that is used to map into a texture image when texture mapping is enabled. The texture coordinates are part of a three-dimensional homogeneous coordinate system (s,t,r,q). Applications often only assign the 2D s and t coordinates, but OpenGL treats this as a special case of the more general 3D homogeneous texture coordinate space. The r and q texture coordinates are vital to techniques that utilize volumetric and projective texturing. When t, r, or q are not explicitly assigned a

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits
ALPHA4	ALPHA				4		
ALPHA8	ALPHA				8		
ALPHA12	ALPHA				12		
ALPHA16	ALPHA				16		
LUMINANCE4	LUMINANCE					4	
LUMINANCE8	LUMINANCE					8	
LUMINANCE12	LUMINANCE					12	
LUMINANCE16	LUMINANCE					16	
LUMINANCE4_ALPHA4	LUMINANCE_ALPHA				4	4	
LUMINANCE6_ALPHA2	LUMINANCE_ALPHA				2	6	
LUMINANCE8_ALPHA8	LUMINANCE_ALPHA				8	8	
LUMINANCE12_ALPHA4	LUMINANCE_ALPHA				12	4	
LUMINANCE16_ALPHA16	LUMINANCE_ALPHA				16	16	
INTENSITY4	INTENSITY						4
INTENSITY8	INTENSITY						8
INTENSITY12	INTENSITY						12
INTENSITY16	INTENSITY						16
R3_G3_B2	RGB	3	3	2			
RGB4	RGB	4	4	4			
RGB5	RGB	5	5	5			
RGB8	RGB	8	8	8			
RGB10	RGB	10	10	10			
RGB12	RGB	12	12	12			
RGB16	RGB	16	16	16			
RGBA2	RGBA	2	2	2	2		
RGBA4	RGBA	4	4	4	4		
RGB5_A1	RGBA	5	5	5	1		
RGBA8	RGBA	8	8	8	8		
RGB10_A2	RGBA	10	10	10	2		
RGBA12	RGBA	12	12	12	12		
RGBA16	RGBA	16	16	16	16		

Table 1: OpenGL Internal Texture Formats. Each internal texture format has a corresponding base internal format and its *desired* component resolutions.

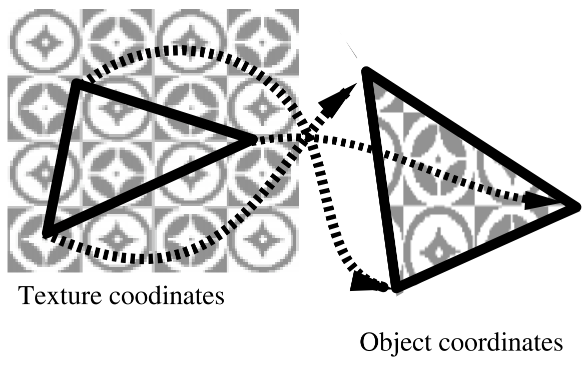


Figure 27. Vertices with Texture Coordinates. Texture coordinates determine how texels in the texture are mapped to the surface of a triangle in object space.

value (as when `glTexCoord1f` is called), their assumed values are 0, 0, and 1 respectively. If the concept of 3D homogeneous texture coordinates is unfamiliar to your, the topic will be revisited in Section 6.16.

OpenGL's interpolation of texture coordinates across a primitive compensates for the appearance of a textured surface when viewed in perspective. While so-called *perspective correct* texture coordinate interpolation is more expensive, failing to account for perspective results in incorrect and unsightly distortion of the texture image across the textured primitive's surface.

Each texture coordinate is assumed to be floating-point value. Each set of texture coordinates must be mapped to a position within the texture image. The coordinates of the texture map range from [0..1] in each dimension. OpenGL can treat coordinate values outside the range [0,1] in one of two ways: clamp or repeat. In the case of clamp, the coordinates are simply clamped to [0,1] causing the edge values of the texture to be stretched across the remaining parts of the polygon. In the case of repeat the integer part of the coordinate is discarded so the texture image becomes an infinitely repeated tile pattern. In the case of clamping, proper filtering may require accounting for border texels or, when no border is specified, the texture border color. OpenGL 1.2 adds a variation on clamping known as *clamp to edge* that clamps such that the border is never sampled.[2] The filtered color value that results from texturing can be used to modify the original surface color value in one of several ways as determined by the *texture environment*. The simplest way replaces the surface color with texel color, either by modulating a white polygon or simply replacing the color value. Simple replacement was added as an extension by some vendors to OpenGL 1.0 and is now part of OpenGL 1.1.

Assigning Texture Coordinates A common question is how texture coordinates are assigned to the vertices of an object. There is no single answer. Sometimes the texture coordinates are some mathematical function of the object coordinates. In other cases, the texture coordinates are manually assigned by the artist that created a given 3D model. Most common 3D object file formats such as VRML or the Wavefront OBJ format contain accompanying texture coordinates. Keep in mind that the assignment of texture coordinates for a particular 3D model is not something that can be done independent of the intended texture to be mapped onto the object.

Optimizing Texture Coordinate Assignment Sloan, Weinstein, and Brederson [90] have explored optimizing the assignment of texture coordinates based on an "importance map" that can encode both intrinsic texture proper-

[2]The *clamp to edge* functionality is also available through the `SGIS_texture_edge_clamp` extension.

ties as well as user-guided highlights. Such importance driven texture coordinate optimization techniques highlight the fact that textured detail is very likely not uniformly distributed for a particular texture image and a particular texture coordinate assignment. Warping the texture image and changing the texture coordinate assignment provides opportunities for improving texture appearance without increasing the texture size.

6.1.3 Texture Coordinate Generation and Transformation

An alternative to assigning texture coordinate explicitly to every vertex is to have OpenGL generate texture coordinates for you. OpenGL's texture coordinate generation (often called *texgen* for short) can generate texture coordinates automatically as a linear function of the eye-space or object-space coordinates or using a special sphere map formula designed for environment mapping.

OpenGL also provides a 4 by 4 *texture matrix* that can be used to transform the per-vertex texture coordinates, whether supplied explicitly or implicitly through texture coordinate generation. The texture matrix provides a means to rescale, translate, or even project texture coordinates before the texture is applied during rasterization.

6.1.4 Filtering

The texture image is a discrete array of texels, but the texture coordinates vary continuously (at least conceptually). This creates a sampling problem. In addition, a fragment can really be thought of as covering some region of the texture image (the fragment's footprint). Filtering also tries to account for a fragment's footprint within the texture image.

OpenGL provides a number of filtering methods to compute the texel value. There are separate filters for magnification (many pixel fragment values map to one texel value) and minification (many texel values map to one pixel fragment). The simplest of the filters is point sampling, in which the texel value nearest the texture coordinates is selected. Point sampling seldom gives satisfactory results, so most applications choose some filter which interpolates. For magnification, OpenGL only supports linear interpolation between four texel values. For minification, OpenGL supports various types of mipmapping [103], with the most useful (and computationally expensive) being tri-linear mipmapping (four samples taken from each of the nearest two mipmap levels and then interpolating the two sets of samples). Some vendors have also added an extension called `SGIS_texture_filter4` that provides a larger filter kernel in which the weighted sum of a 4x4 array of texels is used.

With mipmapping, a texture consists of multiple levels-of-detail (LODs). Each mipmap level is a distinct texture image. The base mipmap level has the highest resolution and is called mipmap level zero. Each subsequent level is half the dimensions (height, width, and depth) until each dimension goes to one and finally all the dimensions reduce to one. For mipmap filtering to work reasonably, each subsequent mipmap level is down-sampled version of the previous mipmap level texture image. Figure 28 shows how texture mipmap levels provide multiple LODs for a base texture image. OpenGL does not provide any built-in commands for generating mipmaps, but the GLU provides some simple routines (`gluBuild1DMipmaps`, `gluBuild2DMipmaps`, and `gluBuild3DMipmaps`[3]) for generating mipmaps using a simple box filter.

During texturing, OpenGL automatically computes (or more likely, approximates) each fragment's LOD parameter based on the partial derivatives of the primitive's mapping of texture coordinates to window coordinates. This LOD parameter is often called lambda (λ). The integer portion of the lambda value determines which mipmap levels to use for mipmap filtering and the fractional portion of the lambda value determines the weighting for selecting or blending mipmaps levels. Because OpenGL handles mipmapping automatically, the details of LOD computation are most interesting to OpenGL implementors, but it is important that users of OpenGL understand the interpolation math so that they will not be surprised by unexpected results.

Additional Control of Texture Level of Detail In OpenGL 1.0 and 1.1, all the mipmap levels of a texture must be specified and consistent. To be consistent, every mipmap level of a texture must be half the dimensions (until

[3]Introduced in GLU version 1.3.

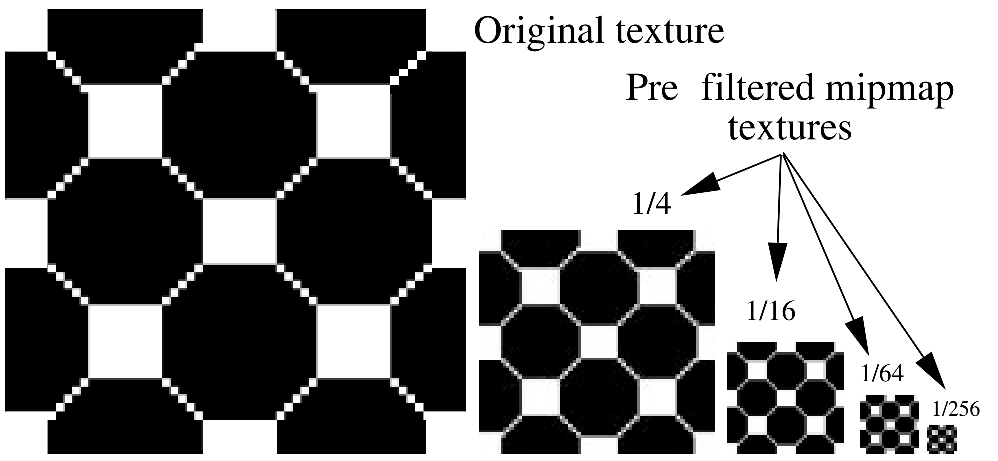


Figure 28. Multiple Levels of Texture Detail using Mipmaps

reaching a dimension of one and excluding border texels) of the previous mipmap LOD, and all the mipmaps must shared the same internal format and borders.

If mipmap filtering is requested for a texture, but all the mipmap levels of a texture are not present or not consistent, OpenGL silently disables texturing. A common pitfall for OpenGL programmers is supplying an inconsistent or incomplete set of mipmap levels and then wondering why texturing does not work. Be sure to specify all the mipmap levels of a texture consistently. If you use the GLU routines for building mipmaps, this is guaranteed.

OpenGL 1.2 relaxes the texture consistency requirement by allowing the application to specify a contiguous range of mipmap levels that must be consistent. This permits an application to still use mipmapping if only the 1x1 through 256x256 mipmap levels of a texture with a 1024x1024 level 0 texture, but not supply the 512x512 and 1024x1024 levels by managing the texture's `GL_TEXTURE_BASE_LEVEL` and `GL_TEXTURE_MAX_LEVEL` parameters. If an application is designed to guarantee a constant frame-rate, one reason the application might constrain the base and maximum LODs in this way is that the application does not have the time to read the 512x512 and 1024x1024 mipmap levels from disk. In this case, the application makes the choice to settle for lower resolution LODs, possibly resulting in blurry textured surfaces, rather than of dropping a frame. Hopefully on subsequent frames, the application can manage to load the full set of mipmap levels for the texture and continue with full texture quality. The OpenGL implementation implements this feature by simply clamping the λ LOD value to the range of available mipmap levels.

Additionally, even when all the mipmap levels are present and consistent, some of the texture images for some levels may be out-of-date if the texture is being dynamically updated using subtexture loads. OpenGL 1.2's `GL_TEXTURE_MIN_LOD` and `GL_TEXTURE_MAX_LOD` texture parameters provide a further means to clamp the λ LOD value to a contiguous range of mipmap levels.[4] Section 6.8 applies this functionality to the task of texture paging.

6.1.5 Texture Environment

The process by which the final fragment color value is derived is called the texture environment function (`glTexEnv`) Several methods exist for computing the final color, each capable of producing a particular effect. One of the most commonly used is the `GL_MODULATE` environment function. The modulate function multiplies or modulates the original fragment color with the texel color. Typically, applications generate polygons with per-vertex lighting

[4]This same functionality for controlling texture level of detail is also available through the `SGIS_texture_lod` extension.

enabled and then modulate the texture image with the fragment's interpolated lit color value to produce a lit, textured surface. The `GL_REPLACE` texture environment[5] is even simpler. The replace function simply replaces the fragment's color with the color from the texture. The same effect as replace can be accomplished in OpenGL 1.1 by using the modulate environment with a constant white current color, though the replace function has a lower computational cost.

The `GL_DECAL` environment function performs simple alpha-blending between the fragment color and an RGBA texture; for RGB textures it simply replaces the fragment color. Decal mode is undefined for other texture formats (luminance, alpha, intensity). The `GL_BLEND` environment function uses the texture value to control the mix of the incoming fragment color and a constant texture environment color.

At the time of this writing, efforts are underway to standardize extensions that enhance the texture environment by adding new functions. For example, there should be a way to add the texture color to the fragment color.

6.1.6 Texture Objects

Most texture mapping applications switch among many different textures during the course of rendering a scene. To facilitate efficient switching among multiple textures and to facilitate texture management, OpenGL uses *texture objects* to maintain texture state.

The state of a texture object consists of the set of texture images for the all mipmap levels of the texture and the texturing parameters such as the texture wrap and minification and magnification filtering modes. Other OpenGL texture-related state such as the texture environment or texture coordinate generation modes are *not* part of a texture object's state. Conceptually, the state of a texture object is just the texture image and the parameters that determine how to filter that image.

As with display lists, each texture object is identified by a 32-bit unsigned integer that serves as the texture's name. Also as with display lists names, the application is free to assign arbitrary unused names to new texture objects. The command `glGenTextures` assists in the assignment of texture object names by returning a set of names guaranteed to be unused. A texture object is bound, prioritized, checked for residency, and deleted by its name. The value zero is reserved to name the default texture of each texture target type. Each texture object has its own texture target type. The three supported texture targets are:

- `GL_TEXTURE_1D`
- `GL_TEXTURE_2D`
- `GL_TEXTURE_3D`

Calling `glBindTexture` binds the named texture object as the current texture for the specified texture target. Instead of creating a texture object explicitly, a texture object is created whenever a texture image or parameter is set for an unused texture object name. Once created a texture object's target (1D, 2D, or 3D) is fixed until the texture object is deleted.

The `glTexImage`, `glTexParameter`, `glGetTexParameter`, `glGetTexLevelParameter`, and `glGetTexImage` commands update or query the state of the currently bound texture of the specified target type. Keep in mind that there are really three current textures, one for each texture target type: 1D, 2D, and 3D. When texturing is fully enabled, the current texture object (i.e., current for the enabled texture target) is used for texturing. When rendering objects with different textures, `glBindTexture` is the way to switch among the available textures.

Keep mind that switching textures is a fairly expensive operation. If a texture is not already resident in dedicated texture memory, switching to a non-resident texture requires that the texture be downloaded to the hardware before use. Even if the texture is already downloaded, caches that maximize texture performance may be invalidated when switching textures. The details of switching textures varies depending on your OpenGL implementation,

[5]Introduced by OpenGL 1.1.

but suffice it to say that OpenGL implementations are inevitably optimized to maximize texturing performance for whatever texture is currently bound so changing textures is something to minimize. Real-world applications often derive significant performance gains by sorting by texture the objects that they render to minimize the number of `glBindTexture` commands required to render the scene. For example, if a scene uses three different tree textures to draw several dozen trees within a scene, it is a good idea to draw all the trees that share a single texture first before switching to a different tree texture.

Texture objects were introduced by OpenGL 1.1. The original OpenGL 1.0 specification did not support texture objects. The thinking at the time was that display lists containing a complete set of texture images and texture parameters could provide a sufficient mechanism for fast texture switches. But display listed textures proved inadequate for several reasons. Recognizing textures embedded in display list efficiently proved difficult. One problem was that a display listed `glTexImage2D` must encapsulate the original image, which might not be the final texture as transformed by the pixel transfer pipeline. Changes to the pixel transfer pipeline state could change the texture image downloaded in subsequent calls of the display list. Unless every pixel transfer state setting was explicitly set in the display list, OpenGL implementations had to maintain the original texture data and be prepared to re-transform it by the current pixel transfer pipeline state when the texture display list is called. Moreover, even if every pixel transfer state setting is explicitly set in the display list, supporting future extensions that add new pixel transfer state would invalidate the optimization. Texture objects store the *post*-pixel transfer pipeline image so texture objects have no such problem. Another issue is that because display lists are not editable, display lists precluded support for subtexture loads as provided by the `glTexSubImage2D` command. Lastly, display lists lack a means to query and update the priority and residency of textures.[6]

6.2 Multitexture

Multitexture refers to the ability to apply two or more distinct textures to a single fragment. Each texture has the ability to supply its own texture color to rasterized fragments. Without multitexture, there is only a single supported texture unit. OpenGL's multitexture support requires that every texture unit be fully functional and maintain state that is independent of any other texture units. Each texture unit has its own texture coordinate generation state, texture matrix state, texture enable state, and texture environment state. However, each texture unit within an OpenGL context shares the same set of texture objects.

Rendering algorithms that require multiple rendering passes can often be reimplemented to use multitexture in operate in less rendering passes. Some effects are only viable with multitexture.

Many OpenGL games such as Quake and Unreal use light maps to improve the lighting quality within their scenes. Without multitexture, light map textures must be modulated into the scene with a second blended rendering pass in addition to a first pass to render the base surface texture. With multitexture, the light maps and base surface texture can be rendered in a single rendering pass. This can cut the transformation overhead almost in half when rendering light maps because a single multitexture rendering pass means that polygons need to only be transformed once. The framebuffer update overhead is also lower when using multitexture to render light maps. When multitexture is used, the overhead of blending in the second rendering pass is completely eliminated. A single multitextured rendering pass can render both the surface texture and the light map texture without any framebuffer blending because the modulation of the surface texture with the light map texture occurs as part of the multitexture texture environment. Light maps are described in more detail in Section 10.2.

The OpenGL 1.2.1 revision of the OpenGL specification [88] includes an Appendix F that introduces the concept of OpenGL Architecture Review Board (ARB) approved extensions and specifies the `ARB_multitexture` extension, the first distinct ARB extension. The original OpenGL 1.2 specification includes an ARB extension called the `ARB_imaging` extension, but the `ARB_imaging` description is intermingled with the core OpenGL 1.2 specification. The `ARB_multitexture` extension is the first ARB extension that is specified in an Appendix distinct from the core specification. The purpose of ARB extensions is to add important new functionality to OpenGL in

[6]While the `SGIX_list_priority` extension does provide a way to prioritize display lists, the concept of querying texture residency, while important to texture objects, is not applicable to display lists.

a modular way that makes it easier and quicker for OpenGL implementors to make available standard OpenGL feature subsets because an ARB extension does not require a complete update of the core OpenGL specification.

6.2.1 Multitexture API Overview

An OpenGL implementation that supports `ARB_multitexture` supports a new set of API routines for controlling the multitexture state of one or more texture units. The `glActiveTextureARB` routine controls which texture unit the existing OpenGL texture commands affect. For example, to enable 2D texturing on texture unit 0 and 1D texturing on texture unit 1, make the following OpenGL calls:

```
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_1D);
```

Note that the state of each texture unit is completely independent. When multitexture is supported, other texture command such as `glTexGen`, `glTexImage2D`, and `glTexParameter` affect the current active texture unit as last set by `glActiveTextureARB`. Other commands such as `glDisable`, `glGetIntegerv`, `glMatrixMode`, `glPushMatrix`, and `glPopMatrix`, also abide by the current active texture unit when updating or querying texture state.

The number of texture units supported can be queried. Indeed, the `ARB_multitexture` specification unfortunately permits an implementation to claim to support the extension but only support a single texture unit. This means that, to be safe, even if you only need a two texture units, you should be careful to query the implementation-dependent constant `GL_MAX_TEXTURE_UNITS_ARB` using `glGetIntegerv`. At the time of this writing, most existing `ARB_multitexture` implementations support only two texture units, but the extension has set aside enumerants for as many as 32 texture units.

Without multitexture, OpenGL supports just a single set of texture coordinates, But with multitexture, each vertex has a number of texture coordinate sets equal to the maximum number of texture units supported by the implementation. The `glMultiTexCoordARB` routines make it easy to supply different texture coordinates for each texture unit. For example:

```
glMultiTexCoord2fARB(GL_TEXTURE0_ARB, s0, t0);
glMultiTexCoord4fARB(GL_TEXTURE1_ARB, s1, t1, r1, q1);
glMultiTexCoord1iARB(GL_TEXTURE2_ARB, s2);
glVertex3f(x, y, z);
```

The behavior of the `glTexCoord` family of routines is specified to update just texture unit zero.

The multitexture extension supports vertex arrays for multiple texture coordinate sets. Because vertex arrays are considered client state, the `glClientActiveTextureARB` command controls which vertex array texture coordinate set that the `glTexCoordPointer`, `glEnableClientState`, `glDisableClientState`, and `glGet-Pointerv` commands affect or query. For example:

```
glClientActiveTextureARB(GL_TEXTURE0_ARB);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
glClientActiveTextureARB(GL_TEXTURE1_ARB);
glTexCoordPointer(2, GL_FLOAT, 0, tex_array_ptr);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

The current raster position has been extended to maintain a distinct texture coordinate set for each supported texture unit.

To simplify the multitexture extension, OpenGL's evaluator and feedback functionality are not extended to support multiple texture coordinate sets. Evaluators and feedback utilize only texture coordinate set 0.

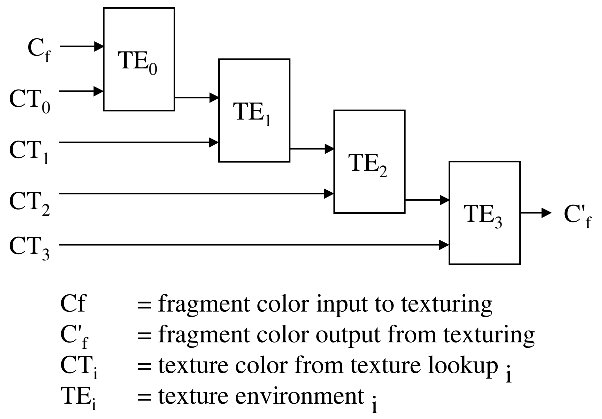

C_f
CT_0
CT_1
CT_2
CT_3
TE_0
TE_1
TE_2
TE_3
C'_f
Cf = fragment color input to texturing
C'_f = fragment color output from texturing
CT_i = texture color from texture lookup $_i$
TE_i = texture environment $_i$

Figure 29. Multitexture Texture Environments. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation. The input fragment color is successively combined with each texture according to the state of the corresponding texture environment, and the resulting fragment color passed as input to the next texture unit in the pipeline.

The `glPushAttrib`, `glPopAttrib`, `glPushClientAttrib`, and `glPopClientAttrib` push and pop respectively all the respective server or client texture state of all texture units when texture-related state is pushed or popped.

6.2.2 Multitexture Texture Environments

The `EXT_multitexture` extension uses a cascade model to combine the incoming fragment color with the fragment's texture contribution from each texture unit. The first enabled texture unit receives the incoming fragment color and applies the texture unit's environment to combine the incoming fragment with the fragment's corresponding texture color for the texture unit. The resulting color is passed to the next enabled texture unit and that texture unit's texture environment is applied using the fragment's corresponding texture color for the texture unit. This process continues until all the enabled texture units are used. The order that the texture environments are applied is the numerical order of the texture units. Figure 29 shows the multitexture texture environment dataflow.

For now, the `EXT_multitexture` texture environment is sufficient to support today's multitexture applications such as light maps or dual-paraboloid environment maps. However, most existing multitexture hardware, and almost certainly all multitexture hardware being designed currently, supports a substantially more flexible facility for combining multiple textures (though the exact details vary from hardware vendor to vendor [15]). As of the writing, work is on-going to standardize a more powerful multitexture texture environment for OpenGL.

6.3 Merging Textures with Specular Highlights

Unfortunately, when a lit polygon includes a specular highlight, the resulting modulated texture will not look correct since the specular highlight simply changes the brightness of the texture at that point rather than the desired effect of adding in some specular illumination. Some vendors have addressed this problem with extensions to perform specular lighting after texturing. The `EXT_separate_specular_color` extension permits the color result from the per-vertex specular lighting computation to be separately interpolated during rasterization and then added to the fragment's color *after* the texture environment. This new step in the OpenGL pipeline is called the color sum. OpenGL 1.2 integrates the separate specular color functionality into the OpenGL core standard. Enabling the separate specular color functionality in OpenGL 1.2 is easy:

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
```

OpenGL's default behavior is restored by:

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SINGLE_COLOR);
```

One limitation of the separate specular color functionality is that the secondary color (as it is called) added in during the color sum stage can *only* be supplied through OpenGL's per-vertex specular lighting computation. When lighting is disabled, there is no way to control the secondary color. The `EXT_secondary_color` extension permits the secondary color to be pass directly into OpenGL via the command `glSecondaryColor3fEXT`. This can be very useful is you wish to perform your own per-vertex lighting model that supplies independent specular highlights. Here is an example assuming the extension is supported:

```
glDisable(GL_LIGHTING);
glEnable(GL_COLOR_SUM_EXT);       /* enable color sum
                                     with explicit secondary color  */
glColor4f(0, 0.3, 0.5);           /* explicit primary color (RGBA)  */
glSecondaryColor3fEXT(1, 1, 0.8); /* explicit secondary color (RGB) */
```

Neither of these techniques address a problem fundamental to per-vertex computation of specular highlights though. Because specular highlights change rapidly over a shiny surface, when an object is not sufficiently tessellated, the specular highlight tend to "wobble" as the object moves or rotates. This is due to the specular highlight sometimes being correctly sampled when a vertex is close to the highlight, but poorly sampled if a vertex is not close to the highlight. Other techniques that can be used to address this problem are discussed in Section 10.1.1.

6.4 Texture Borders and Tiling

A useful (and sometimes misunderstood) feature of OpenGL is the texture border. OpenGL supports either a constant texture border color or a border that is a portion of the edge of the texture image. The key to understanding texture borders is understanding how textures are sampled when the texture coordinate values are near the edges of the [0,1] range and the texture wrap mode is set to `GL_CLAMP`. For point sampled filters, the computation is quite simple: the border is never sampled. However, when the texture filter is linear and the texture coordinate reaches the extremes (0.0 or 1.0), the resulting texel value is a 50% mix of the border color and the outer texel of the texture image at that edge (25% and 75% at the corners).

The texture border is most useful when attempting to use a single high resolution texture image which is too large for the OpenGL implementation to support as a single texture map. For this case, the texture can be broken up into multiple tiles, each with a 1 pixel wide border from the neighboring tiles. The texture tiles can then be loaded and used for rendering in several passes. For example, if a 1K by 1K texture is broken up into four 512 by 512 images, the four images would correspond to the texture coordinate ranges (0-0.5,0-0.5), (0.5,1.0,0-0.5), (0-0.5,0.5,1.0) and (.5-1.0,.5-1.0). As each tile is loaded, only the portions of the geometry that correspond to the appropriate texture coordinate ranges for a given tile should be drawn. If you had a single triangle whose texture coordinates were (.1,.1), (.1,.7), and (.8,.8), you would clip the triangle against the four tile regions and draw only the portion of the triangle that intersects with that region as shown in Figure 30. At the same time, the original texture coordinates need to be adjusted to correspond to the scaled and translated texture space represented by the tile. This transformation can be easily performed by loading the appropriate scale and translation onto the texture matrix stack.

Unfortunately, OpenGL does not provide much assistance for performing the clipping operation. If the input primitives are quads and they are appropriately aligned in object space with the texture, then the clipping operation

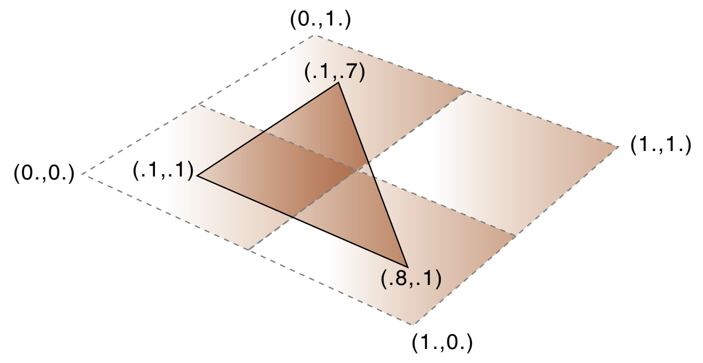


Figure 30. Texture Tiling

is trivial; otherwise, it may involve substantially more work. One method to assist with the clipping uses stenciling to control which textured fragments are kept. Stencil testing is described in Section 8.6. Then you are left with the problem of setting the stencil bits appropriately. The easiest way to do this is to produce alpha values that are proportional to the texture coordinate values and use `glAlphaFunc` to reject alpha values that you do not wish to keep. Unfortunately, you can not easily map a multidimensional texture coordinate value (e.g., s,t) to an alpha value by simply interpolating the original vertex alpha values, so it is best to use a multidimensional texture itself which has some portion of the texture with zero alpha and some portion with it equal to one. The texture coordinates are then scaled so that the textured polygon map to texels with an alpha of 1.0 for pixels to be retained and 0.0 for pixels to be rejected.

OpenGL 1.2 adds an alternative clamping behavior when the texture wrap mode is set to `GL_CLAMP_TO_EDGE`.[7] This mode clamps the texture coordinates such that the texture border is never sampled. The filtered color is derived only from texels at the edge of the texture image. Unlike OpenGL's standard texture clamping, the clamp to edge behavior is unable to guarantee a consistent border appearance when used with mipmapping because the clamping range changes with each mipmap level because the clamping range depends on the selected mipmap level's dimensions. The clamp to edge behavior is easier to implement in hardware than OpenGL's standard clamping with texture borders because the texture dimensions are not augmented by additional border texels the dimensions are efficient powers of two. The clamp to edge behavior matches the texture clamping behavior of Direct3D.

6.5 Mipmap Generation

Having explored the possibility of tiling low resolution textures to achieve the effect of high resolution textures, we now examine methods for generating better texturing results without resorting to tiling. Again, OpenGL supports a modest collection of filtering algorithms, the highest quality of the minification algorithms being `GL_LINEAR_MIPMAP_LINEAR`. OpenGL does not specify a method for generating the individual mipmap levels (LODs). Each level can be loaded individually, so it is possible, but probably not desirable, to use a different filtering algorithm to generate each mipmap level.

The GLU library provides a very simple interface (`gluBuild2DMipmaps`) for generating all of the 2D levels required. The algorithm currently employed by most implementations is a box filter. There are a number of advantages to using the box filter; it is simple, efficient, and can be repeatedly applied to the current level to generate the next level without introducing filtering errors. However, the box filter has a number of limitations that

[7]The same functionality is also available through the `SGIS_texture_edge` extension.

can be quite noticeable with certain textures. For example, if a texture contains very narrow features (e.g., lines), then aliasing artifacts may be very pronounced.

The best choice of filter functions for generating mipmap levels is somewhat dependent on the manner in which the texture will be used and it is also somewhat subjective. Some possibilities include using a linear filter (sum of four pixels with weights [1/8,3/8,3/8,1/8]) or a cubic filter (weighted sum of eight pixels). Mitchell and Netravali [64] propose a family of cubic filters for general image reconstruction which can be used for mipmap generation. The advantage of the cubic filter over the box is that it can have negative side lobes (weights) which help maintain sharpness while reducing the image. This can help reduce some of the blurring effect of filtering with mipmaps.

When attempting to use a filtering algorithm other than the one supplied by the GLU library, it is important to keep a couple of things in mind. The highest resolution (finest) image of the mipmap (LOD 0) should always be used as the input image source for each level to be generated. For the box filter, the correct result is generated when the preceding level is used as the input image for generating the next level, but this is not true for other filter functions. Each time a new (coarser) level is generated, the filter needs to be scaled to twice the width of the previous version of the filter. A second consideration is that in order to maintain a strict factor of two reduction, filters with widths wider than two need to sample outside the boundaries of the image. This is commonly handled by using the value for the nearest edge pixel when sampling outside the image. However, a more correct algorithm can be selected depending on whether the image is to be used in a texture in which a repeat or clamp wrap mode is to be used. In the case of repeat, requests for pixels outside the image should wrap around to the appropriate pixel counted from the opposite edge, effectively repeating the image.

Mipmaps may be generated using the host processor or using the OpenGL pipeline to perform some of the filtering operations. For example, the `GL_LINEAR` minification filter can be used to draw an image of exactly half the width and height of an image which has been loaded into texture memory, by drawing a quadrilateral with the appropriate transformation (i.e., the quad projects to a rectangle one fourth the area of the original image). This effectively filters the image with a box filter. The resulting image can then be read from the color buffer back to host memory for later use as LOD 1. This process can be repeated using the newly generated mipmap level to produce the next level and so on until the coarsest level has been generated.

The above scheme seems a little cumbersome since each generated mipmap level needs to be read back to the host and then loaded into texture memory before it can be used to create the next level. The `glCopyTexImage` capability, added in OpenGL 1.1, allows an image in the color buffer to be copied directly to texture memory.

The texture LOD extension to OpenGL 1.1 (standard in OpenGL 1.2) can be used by an application to use the previously computed mipmap level in the target texture itself as the texture applied to compute the final texture level. In this way, the application would never need to change texture objects.

The above method outlines an algorithm for generating mipmap levels using the existing texture filters. There are other mechanisms within the OpenGL pipeline that can be combined to do the filtering. Convolution can be implemented using the accumulation buffer (this will be discussed in more detail in Section 13.3.3. A texture image can be drawn using a point sampling filter (`GL_NEAREST`) and the result added to the accumulation buffer with the appropriate weighting. Different pixels (texels) from an NxN pattern can be selected from the texture by drawing a quad that projects to a region 1/N x 1/N of the original texture width and height with a slight offset in the s and t coordinates to control the nearest sampling. Each time a textured quad is rendered to the color buffer it is accumulated with the appropriate weight in the accumulation buffer. Combining point-sampled texturing with the accumulation buffer allows the implementation of nearly arbitrary filter kernels. Sampling outside the image however still remains a difficulty for wide filter kernels. If the outside samples are generated by wrapping to the opposite edge, then the `GL_REPEAT` wrap mode can be used.

6.6 Texture Map Limits

In addition to issues concerning the maximum texture resolution and the methods used for generating texture images there are also some pragmatic details with using texturing. Many OpenGL implementations hardware accelerate texture mapping and have finite storage for texture maps being used. Many implementations will

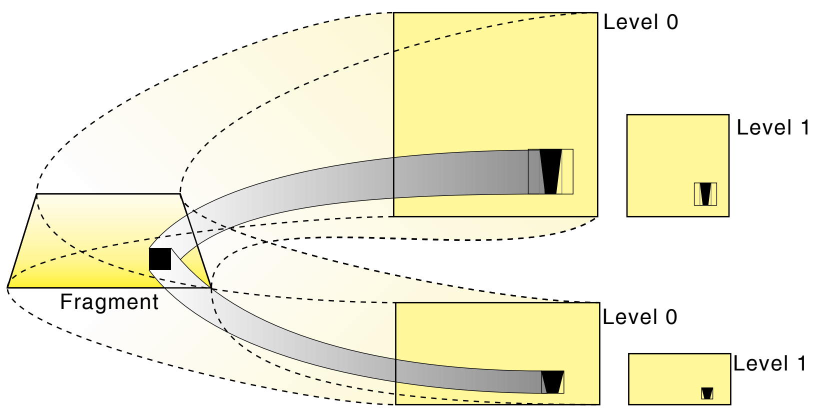


Figure 31. Footprint in Anisotropically Scaled Texture

virtualize this resource so that an arbitrarily large set of texture maps can be supported within an application, but as the resource becomes oversubscribed performance will degrade. In applications that need to use multiple texture maps there is a tension between the available storage resources and the desire for improved image quality.

This simply means that it is unlikely that every texture map can have an arbitrarily high resolution and still fit within the storage constraints; therefore, applications need to anticipate how textures will be used in scenes to determine the appropriate resolution to use. Note that texture maps need not be square; if a texture is typically used with an object that is projected to a non-square aspect ratio then the aspect ratio of the texture can be scaled appropriately to make more efficient use of the available storage.

6.7 Anisotropic Texture Filtering

Currently, OpenGL only provides an isotropic filter for texture minification. This means that the amount of filtering done along the s and t axes of the texture is the same, and is the maximum of the filtering needed along each of the two axes individually. This can lead to excessive blurring when a texture is viewed at any angle other than straight on. If it is known that a texture will always be viewed at a given angle or range of angles, it can be created in a way that reduces over-filtering.

Suppose a textured square is rendered as shown in the left of Figure 31. The texture is shown in the right. Consider the fragment that is shaded dark. Its ideal footprint is shown in the diagram of the texture as the dark inner region. But since the minification filter is isotropic, the actual footprint is forced to a square that encloses the dark region. A mipmap level will be chosen in which this square footprint is properly filtered for the fragment; in other words, a mipmap level will be selected in which the size of this square is closest to the size of the fragment. That mipmap is not level zero but level 1 or higher. Hence, at that fragment more filtering is needed along t than along s, but the same amount of filtering is done in both. The result will be that the texture will be blurred more than it needs to be.

To avoid this problem, do the extra filtering along t when you create the texture, and make the texture have the same width but only half the height. See Figure 31. The footprint now has an aspect ratio that is more square, so the enclosing square is not much larger, and is closer to the size to the fragment. Level 0 will be used instead of a higher level. Another way to think about this is that by using a texture that is shorter along t, you reduce the amount of minification that is required along t.

The closer the filtered mipmaps aspect ratio matches the projected aspect ratio of the geometry, the more accurate

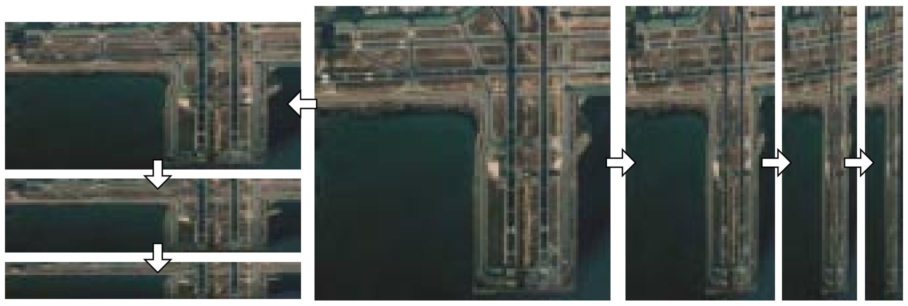


Figure 32. Creating a Set of Anisotropically Filtered Images

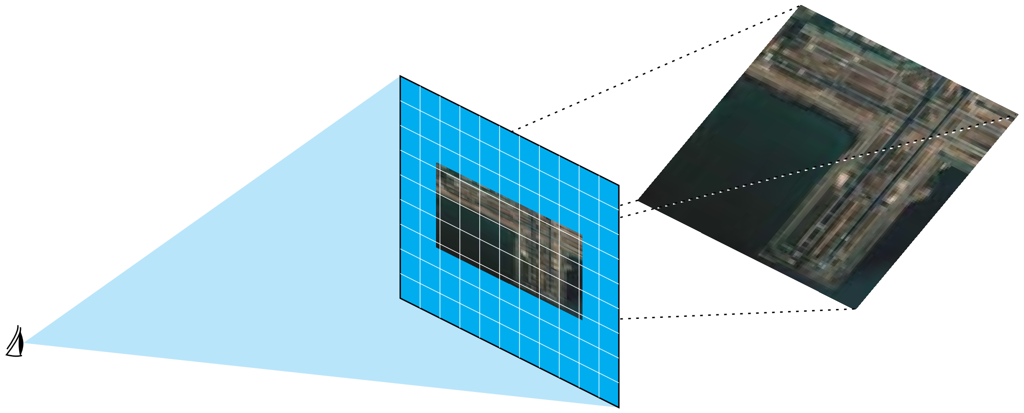


Figure 33. Geometry Orientation and Texture Aspect Ratio

the sampling will be. An application can minimize excessive blurring at the expense of texture memory by creating a set of re-sampled mipmaps with different aspect ratios.

The application can choose the mipmap that most closely corresponds to the texture scaling ratio being applied to the textured terrain. This ratio can be quickly estimated by computing the angle between the viewers line of sight and a plane representing the terrains average orientation. Using texture objects, the application can switch to the mipmap will provide the best results.

1. Re-sample the texture data into different aspect ratios (`gluScaleImage` can be used for this purpose).
2. Create a set of mipmaps corresponding to each image aspect ratio.
3. At each frame, compute the best aspect ratio using the angle between the viewers line of sight and the terrain.
4. Make the mipmap with the best aspect ratio current for texturing the terrain.

Since texture levels must have power of two dimensions, it would appear that the only aspect ratios that can be prefiltered are 1:4, 1:2, 1:1, 2:1, 4:1, etc. You can actually define smaller aspect ratio step size by using a

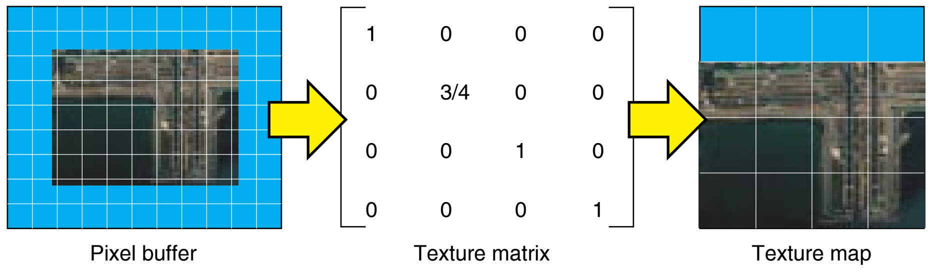


Figure 34. Non Power-of-2 Aspect Ratio Using Texture Matrix

combination of incomplete texture images and use of the texture transform matrix. For example, say you want a ratio of 3:4. You cannot define a mipmap with lengths of this ratio, but you can define a 1:1 ratio mipmap and define an image that is scaled into a 3:4 ratio within it. The part of the texture that is not used should be placed along the top (maximum t coordinates) or right (maximum s coordinates) edge of the texture image. The scaled image can be any size, as long as it fits within the texture level. You can then create a mipmap in the normal way.

Using this mipmap for some textured geometry with a 3:4 ratio, results in an incorrect textured image. Be sure to set the texture transform matrix to rescale the narrower side of the texture (in our example in the t direction) by 3/4:

This will change the apparent size ratio between the pixels and textures in the texture filtering system, giving you the proper results. This technique would not work well with a wrapped texture; in our example, there is a discontinuity in the image when you filter outside the range of 0 to 1 in t. However, in our example, wrapping in s would work fine.

6.8 Paging Textures

As applications simulate higher levels of realism, the amount of texture memory they require can increase dramatically. Texture memory is a limited, expensive resource, so loading high resolution images as textures is not always feasible. Applications are often forced to resample their images at a lower resolution to make them fit in texture memory, with a corresponding loss of realism and image quality. If an application must view the entire textured image at high resolution, there may be no alternative to this approach.

But many applications have texture requirements that can be structured so that only a small area of large texture has to be shown at full resolution. For example when textures are used to produce a realistic flight simulation environment, only the textured terrain close to the viewer has to show fine detail; terrain far from the viewer is textured using low resolution texture levels, since a pixel corresponding to these areas covers many texels at once. For many applications that use large texture maps, the maximum amount of texture memory in use for any given viewpoint is bounded.

Applications can take advantage of this phenomena through texture paging. Rather than loading complete levels of a large image, only the portion of the image closest to the viewer is kept in texture memory. The rest of the image is stored in system memory, or on disk. As the viewer moves, the contents of texture memory are updated to keep the closest portion of the image loaded.

There are two different approaches that could be used to address the problem. The first is to subdivide the texture image into fixed sized tiles and selectively draw the geometry that corresponds to each image tile, one at a time, reloading texture memory for each new tile. This approach is difficult to implement. Tile boundaries are a problem for `GL_LINEAR` filters since the locations where the geometry crosses tile boundaries need to be resampled properly.

The problem could be addressed by clipping the geometry so that the texture coordinates are kept within the [0.0, 1.0] range and then use texture borders to handle the edges of each image tile. Clipping geometry to match each image tile itself can be a difficult problem, especially if the geometry is changing dynamically. For example, terrain close to the viewer might be replaced with more highly tessellated geometry to increase realism, while geometry far from the viewer is tessellated more coarsely to improve rendering performance. In general, forcing a correspondence between texture and geometry beyond what is established by texture coordinates is something to be avoided, since it adds additional complication and software quality problems to the application.

A more sophisticated solution is to take advantage of texture coordinate wrapping to page textures without having to tile the textured geometry. To make this clear, consider a single level texture. Define a viewing frustum that limits the amount of visible geometry to a small area, small enough that the visible geometry can be easily textured. Now imagine that the entire texture image is stored in system memory. As the viewer moves, the image in texture memory can be updated so that it exactly corresponds to the geometry visible in the viewing frustum:

1. Given the current view frustum, compute the visible geometry.
2. Set the texture transform matrix to map the visible texture coordinates into 0 to 1 in s and t.
3. Use `glTexImage2D` to load texture memory with the appropriate texel data, using `GL_SKIP_PIXELS` and `GL_SKIP_ROWS` to index to the proper subregion.

This technique would remap the texture coordinates of the visible geometry to match texture memory, then load the matching texture image into texture memory using `glTexImage2D`.

6.8.1 Texture Subloading

While this technique works, it is a very inefficient user of texture bandwidth. Even if the viewer moves a small amount, the entire texture level is reloaded. Performance can be improved by taking advantage of texture subloading.

If the viewer is smoothly traversing textured terrain, you can take advantage of the fact by incrementally updating the contents of texture memory. Instead of completely reloading the contents of texture memory, you can reload the section that has gone out of view from the last frame with the portion of the image that has just come into view this frame. This technique works because of texture coordinate wrapping. When `GL_TEXTURE_WRAP_S` and `GL_TEXTURE_WRAP_T` are set to `GL_REPEAT` (the default), the integer part of texture coordinates are discarded when mapping into texture memory. In effect, texture coordinates the go off the edge of texture memory on one side, "wrap around" to the opposite side. Using subloading, the updating technique looks like this:

1. Given the current and previous view frustum, compute how the range of texture coordinates have changed.
2. Transform the change of texture coordinates into one or more regions of texture memory that need to be updated.
3. Use `glTexSubImage` to update the appropriate regions of texture memory, use `GL_SKIP_PIXELS` and `GL_SKIP_ROWS` to index into the texture image.

If the subloads are computed properly, this technique does not require transforming texture coordinates using the the texture transform matrix. Updating texture memory can take from 1 to 4 subloads.

On many systems, texture subloads can be very inefficient when narrow regions are being loaded. The subloading method can be modified ensure that only subloads above a minimum size are allowed, at the cost of some additional texture memory. The change is simple. Instead of updating every time the view position changes, ignore position changes until the accumulated change requires a subload above the minimum size. Normally this will result in out of date texture data being visible around the edges of the textured geometry. To avoid this, an *invalid region* is specified around the periphery of the texture level, and the view frustum is adjusted so the that geometry textured from the texels from the invalid region are never visible. This technique allows updates to be cached, improving performance.

This paging technique depends on only a limited region of the textured geometry being visible. In this example we are depending on the limits of the view frustum to only allow properly textured geometry to be visible. If the view frustum were expanded, we'd see the texture image wrapping over the surrounding geometry. Even with these limitations, this technique can be expanded to include mipmapped textures.

Since OpenGL does not understand paged mipmaps, the application can not simply define a very large mipmap and not expect the OpenGL implementation to try to allocate the texture memory needed for all the mipmap levels. Instead the application must use the texture LOD control functionality in OpenGL 1.2 (or the `EXT_texture_lod` extension) to define a small number of active levels, using the `GL_TEXTURE_BASE_LEVEL`, `GL_TEXTURE_MAX_LEVEL`, `GL_TEXTURE_MIN_LOD` and `GL_TEXTURE_MAX_LOD` with the `glTexParameter` call. An invalid region must be established and a minimum size update must be set so that all levels can be kept in sync with each other when updated. For example, a subload 32 texels wide at the top level must be accompanied by a subload 16 texels wide at the next coarser level, if mipmapping is going to filter properly. Multiple images at different resolutions will have to be kept in system memory as source images to load texture memory.

If the viewer zooms in or zooms out of the geometry, the texturing system may require levels that are not available in the paged mipmap. The application can avoid this problem by computing the mipmap levels that are needed for any given viewer position, and keeping a set of paged mipmaps available, each representing a different set of LOD levels. The coarsest set could be a normal mipmap, for when the viewer is very far away from the geometry.

6.8.2 Paging Images in System Memory

Up to this point, we've assumed that the texel data is available as a large contiguous image in system memory. Just as texture memory is a limited resource, it also makes sense to conserve system memory as well. For very large texture images, the image data can be divided into tiles, and paged into system memory. This paging can be kept separate from the paging going on from system memory to texture memory. The only difference will be in the offsets required to index the proper region in system memory to download, and increase the number of subloads required to update texture memory. A sophisticated system can wrap texture image data in system memory just as texture coordinates are wrapped in texture memory.

Consider the case of a two dimensional image roam, illustrated in Figure 35, in which the view is moving to the right. As the view pans to the right, new texture tiles must be added to the right edge of the current portion of the texture and old tiles could be discarded from the left edge.

Tiles discarded on the right side of the image create holes where new tiles could be loaded into the texture, but there is a problem with the texture coordinates.

The ability to load subregions within a texture has other uses besides these paging applications. Without this capability textures must be loaded in their entirety and their widths and heights must be powers of two. In the case of video data, the images are typically not powers of two so a texture of the nearest larger power of two can be created and only the relevant subregion needs to be loaded. When drawing geometry, the texture coordinates are simply constrained to the fraction of the texture which is occupied with valid data. Mipmapping can not easily be used with non-power-of-two image data since the coarser levels will contain image data from the invalid region of the texture.

6.8.3 Implementing High Resolution Textured Terrain

Tanner, Migdal, and Jones [94] describe a hardware solution called *clip mapping* for supporting extremely large textures. The approach is implemented in SGI's InfiniteReality graphics subsystem. The basic clip mapping functionality is accessed using the `SGIX_clipmap` extension. In addition to requiring hardware support, the system also requires significant software management of the texture data too, in part, simply due to the massive texture

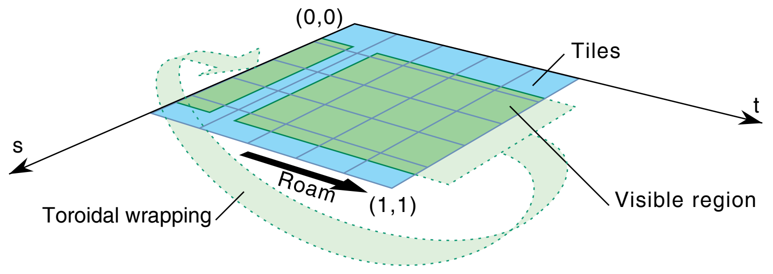


Figure 35. 2D Image Roam

sizes that can be supported. While the clip map approach has no inherent limit to its maximum resolution, the InfiniteReality hardware implementation supports clip map textures to sizes up to 32,768 by 32,768 [65]. The clip map itself is essentially a dynamically updatable partial mipmap. Highest resolution texture data is available only around a particular point in the texture called the *clip center*. To ensure that clip mapped surfaces are shown at the highest possible texture resolution, software is required to dynamically reposition the clip center as necessary. Repositioning the clip center requires partial dynamic updates of the clip map texture data. With software support for repositioning the clip center and managing the off-disk texture loading and caching required, clip mapping offers the opportunity to dynamically roam over and zoom in and out of huge textured regions. The technique has obvious applications for unconstrained viewing of high resolution satellite imagery at real-time rates.

Hüttner [53] describes another approach using only OpenGL's base mipmap functionality to support very high resolution textures. Hüttner proposes a data structure called a *MIPmap pyramid grid* or MP-grid. The MP-grid is essentially a set of mipmap textures arranged in a grid to represent an aggregate high-resolution texture that would be larger than the OpenGL implementation's largest supported texture. For example, a 4 by 4 grid of 1024x1024 mipmapped textures could be used to represent a 4096x4096 aggregate texture. Typically, the aggregate texture is terrain data intended to be draped over a polygonal mesh representing the terrain's geometry. Before rendering, the MP-grid algorithm first classifies each terrain polygon based on which grid cells within the MP-grid the polygon covers. During rendering, each grid cell is considered in sequence. Assuming the grid cell is covered by polygons in the scene, the mipmap texture for the grid cell is bound. Then, all the polygons covering the grid cell are rendered with texturing enabled. Because a polygon may not exist completely within a single grid cell, care must be taken to intersect such polygons with the boundary of all the grid cells that the polygon partially covers.

Hüttner compares the MP-grid scheme to the clip map scheme and notes that the MP-grid approach does not require special hardware and does not depend on determining a single viewer-dependent clip center as needed in the clip map approach. However, the MP-grid approach requires special clipping of the surface terrain mesh to the MP-grid. No such clipping is required when clip mapping. Due its special hardware support, the clip mapping approach is most likely better suited to the very biggest high resolution textures.

6.9 Transparency Mapping and Trimming with Alpha

The alpha component of textures can be used to solve a number of interesting problems. Intricate shapes such as an image of a tree can be stored in texture memory with the alpha component acting as a matte (1 where the image is opaque, 0 where it is transparent, and a fractional value along the edges). When the texture is applied to geometry, blending can be used to composite the image into the color buffer or the alpha test can be used to discard pixels with a 0 alpha component using `GL_EQUALS` test. To maximize performance, set the alpha test to `GL_LESS` and discard pixels with a small alpha value, for example less than .05. This way some more pixels are discarded that do not contribute significantly to the image.

The advantage of using the alpha test instead of alpha blending is that blending typically degrades the performance of fragment processing. With alpha testing fragments with zero alpha are rejected before they get to the color buffer. A disadvantage of alpha testing is that the edges will not be blended into the scene so the edges will not be properly antialiased.

The alpha component of a texture can be used in other ways, for example, to cut holes in polygons or to trim surfaces. An image of the trim region is stored in a texture map and when it is applied to the surface, alpha testing or blending can be used to reject the trimmed region. This method can be useful for trimming complex surfaces in scientific visualization applications.

6.10 Billboards

It is often desirable to replace intricate geometry with simpler texture mapped geometry to increase realism and performance. Billboarding is a technique in which complex objects such as trees are drawn with simple planar texture mapped geometry and the geometry is transformed to face the viewer. The transformation typically consists of a rotation to orient the object towards the viewer and a translation to place the object in the correct position. For the case of the tree, an object with roughly cylindrical symmetry, an axial rotation is used to rotate the geometry for the tree, typically a quadrilateral, about the axis running parallel to the tree trunk.

For the simple case of the viewer looking down the negative z-axis and the up vector equal to the positive y-axis, the angle of rotation can be determined by computing the eye vector from the model view matrix M

$$\vec{V}_{eye} = M^{-1} \begin{pmatrix} 0 \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

and the rotation θ about the y axis is computed as

$$\begin{aligned} \cos\theta &= \vec{V}_{eye} \cdot \vec{V}_{front} \\ \sin\theta &= \vec{V}_{eye} \cdot \vec{V}_{right} \end{aligned}$$

where

$$\begin{aligned} \vec{V}_{front} &= (0,0,1) \\ \vec{V}_{right} &= (1,0,0) \end{aligned}$$

Once θ is computed, construct a rotation matrix R representing this rotation about the y-axis ($\vec{V}_{up}$). Then concatenate this rotation matrix with the modelview matrix to make a combined matrix called MR. Use this combined matrix to transform the billboard geometry.

To handle the more general case of an arbitrary billboard rotation axis, an intermediate alignment rotation A of the billboard axis into the $\vec{V}_{up}$ axis is computed as

$$\begin{aligned} \vec{axis} &= \vec{V}_{up} \times \vec{V}_{billboard} \\ \cos\theta &= \vec{V}_{up} \cdot \vec{V}_{billboard} \\ \sin\theta &= ||\vec{axis}|| \end{aligned}$$

and the matrix transformation is replaced with MAR. Note that the preceding calculations assume that the projection matrix contains no rotational component.

In addition to objects which are cylindrically symmetric, it is also useful to compute transformations for spherically symmetric objects such as smoke, clouds and bushes. Spherical symmetry allows billboards to rotate up and down

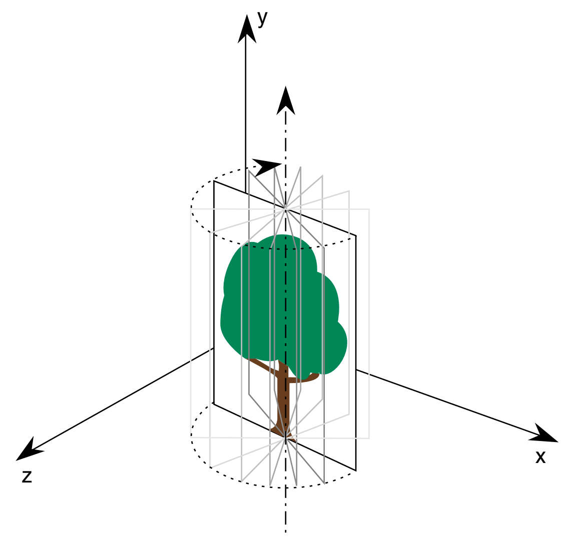


Figure 36. Billboard with Cylindrical Symmetry

as well as left and right, whereas cylindrical behavior only allows rotation to the left or right. Cylindrical behavior is suited to objects such as trees which should not bend backward as the viewer's altitude increases.

Objects which are spherically symmetric are rotated about a point to face the view and thus provide more freedom in computing the rotations. An additional alignment constraint can resolve this freedom. For example, an alignment constraint which keeps the object oriented in a consistent fashion, such as upright. This constraint can be enforced in object coordinates when the objective is to maintain scene realism, perhaps to maintain the orientation of plume of smoke consistently with other objects in a scene. The constraint can also be enforced in eye coordinates which can be used to maintain alignment of an object relative to the screen, for example, keeping annotations such as text aligned horizontally on the screen.

The computations for the spherically symmetric case are a minor extension of the computations for the arbitrarily aligned cylindrical case. First an alignment transformation, A, is computed to rotate the alignment axis onto the up vector followed by a rotation about the up vector to align the face of the billboard with the eye vector. A is computed as

$$\begin{aligned} \vec{axis} &= \vec{V}_{up} \times \vec{V}_{alignment} \\ \cos\theta &= \vec{V}_{up} \cdot \vec{V}_{alignment} \\ \sin\theta &= ||\vec{axis}|| \end{aligned}$$

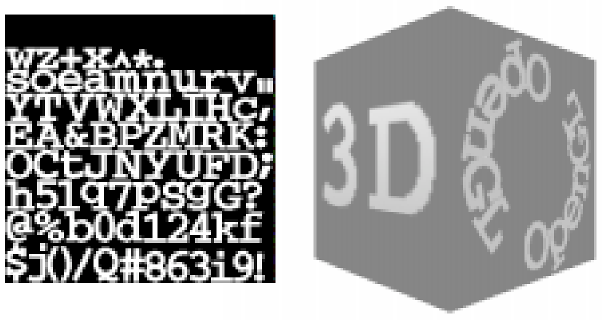


Figure 37. Texture Containing Font Glyphs and Rendering Example

where $\vec{V}_{alignment}$ is the billboard alignment axis with the component in the direction of the eye direction vector removed

$$\vec{V}_{alignment} = \vec{V}_{billboard} - (\vec{V}_{eye} \cdot \vec{V}_{billboard})\vec{V}_{eye}$$

A rotation about the up vector is then computed as for the cylindrical case.

6.11 Rendering Text

A novel use for texturing is rendering antialiased text [41, 57]. Characters are stored in a 2D texture map as for the tree image described above. When a character is to be rendered, a polygon of the desired size is texture mapped with the character image. Since the texture image is filtered as part of the texture mapping process, the quality of the rendered character can be quite good. Text strings can be drawn efficiently by storing an entire character set within a single texture. Rendering a string then becomes rendering a set of quads with the vertex texture coordinates determined by the position of each character in the texture image. Another advantage of this method is that strings of characters may be arbitrarily oriented and positioned in three dimensions by orienting and positioning the polygons.

Figure 37 shows an example of a texture image packed with glyphs for rendering text on the left. The right side of the figure is a rendering example using the texture. Note how each glyphs in the rendering example matches the corresponding glyph in the texture.

The `GL_INTENSITY` texture format is a good texture format for textures containing glyphs because the texture format is compact. The intensity texture format replicates the single intensity component value in the red, green, blue, and alpha channels. When rendering colored glyphs, use the `GL_MODULATE` texture environment and set the current color to the desired glyph color.

Because text is often rendered on planar surfaces, you may need to use stencil testing or polygon offset to avoid depth buffering artifacts caused by the co-planarity of the surface and glyph polygons.

The competing methods for drawing text in OpenGL include bitmaps, vector fonts, and outline fonts rendered as polygons. The texture method is typically faster than bitmaps and comparable to vector and outline fonts. A disadvantage of the texture method is that the texture filtering may make the text appear somewhat blurry. This can be alleviated by taking more care when generating the texture maps (e.g., sharpening them). If mipmaps are constructed with multiple characters stored in the same texture map, care must be taken to ensure that map levels are clamped to the level where the image of a character has been reduced to 1 pixel on a side. Characters should also be spaced far enough apart that the color from one character does not contribute to that of another when filtering the images to produce the levels of detail.

6.12 Texture Mosaicing

The method described above for grouping several images together in a single texture turns out to be useful in other applications as well. In some OpenGL implementations the cost of binding a texture object can limit the overall performance of the application when a large number of textures are being used in each frame. The situation can be mitigated to some extent by packing textures which are used in the same scene together in a single object to reduce the number of texture binds. Also, some images may not need a full power of two for their width or height leaving an opportunity to use texture memory more efficiently if multiple images can be packed together.

Geometry which uses an image within a mosaiced texture has its texture coordinates scaled and biased to index only the texels corresponding to its image. As in the case of character rendering, the individual images in the mosaic must be separated far enough apart so that they do not interfere during filtering. Careful attention should be paid to mipmap generation to ensure that multiple images are not blurred together in a level. The texture LOD clamping capability in OpenGL 1.2 can be used to restrict the range of coarse LODs which are used or mosaiced textures may be constructed from similar enough images that an appropriate single image can be constructed for each level of detail. It may also be useful to pack images together which use the same texture environments to reduce the number of texture environment changes as well.

6.13 Texture Coordinate Generation

Texture coordinates for a fragment are computed by interpolating the texture coordinates for a set of vertices. OpenGL provides several mechanisms for specifying the texture coordinates at each vertex: texture coordinates may be supplied directly by the application using the `glTexCoord` commands or vertex arrays, they may be generated automatically from parametric maps for evaluators, or they may be generated directly by OpenGL using a generation function.

OpenGL supports two mechanisms for computing a texture coordinate directly: distance from a plane, or the reflection vector using the vertex position and normal to compute this vector. The first form is useful for making texture coordinates which are proportional to the distance from the object to some other location and can be computed in either object coordinates or eye coordinates. The latter is useful for environment mapping with a sphere map. The texture coordinate generation function is specified separately for each texture coordinate.

6.14 Color Coding and Contouring

One application for object linear texture coordinate generation is color coding objects by distance. For example, a terrain model can be colored by altitude using a 1D texture map to hold the coloring scheme and specifying a generation function for the s coordinate which measures the distance from the plane $y = 0$. Suppose that the vertex coordinates are specified in meters and distances less than 50 meters are colored blue, distances between 50 and 800 meters green, distances between 800 and 1000 meters white. This means that a 1D texture map is created with the first 5% blue, the next 75% green and the remaining 20% white. A 64 or 128 element texture map provides enough resolution to distinguish between the levels. Specifying `GL_OBJECT_LINEAR` for the texture generation mode and an `GL_OBJECT_PLANE` equation of (0, 1/1000, 0, 0) for the s coordinate will set s to the y value of the vertex scaled by 1/1000.

The same basic technique can be used to draw contour lines on an object, for example, in topography applications to indicate lines of constant elevation. For this example, a 1D texture map is used which is all one color except at regularly spaced intervals (say, every eighth texel) where a tick mark is added in a different color. A coordinate wrap mode of `GL_REPEAT` is used to create repeating lines across the object being contoured. If a `GL_OBJECT_LINEAR` generation function is used then the contours are anchored to the model. If a `GL_EYE_LINEAR` generation function is used then the coordinates are evaluated in eye space and the contours stay fixed in space rather than moving with the object.

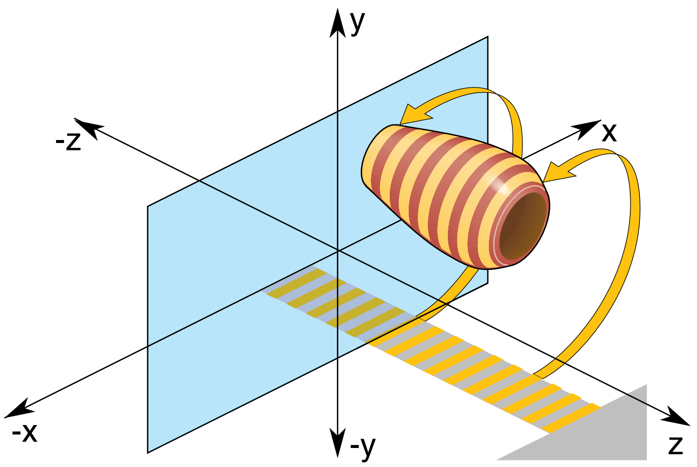


Figure 38. Contour Generation Using TexGen

6.15 Visualizing Surface Orientations

One-dimensional texturing and the texture matrix can be used to visualize the overall orientation of surface normals relative to a specific direction. One unexpected application of visualizing surface orientations is to optimize specific types of manufacturing processes.

Bailey and Clark [7] describe a manufacturing process for the automated fabrication of solid freeform objects. The process makes an abstract object "real" by laminating together sheet after sheet of paper with each sheet stacked on top of the previous sheet. Each sheet of paper corresponds to a thin layer in the object under construction. Before each lamination, a laser first cuts each sheet to the proper shape for its corresponding level within the object. The laser will also cut a crosshatch pattern in the region of the sheet not belonging to the object level. Later, the crosshatched regions will be brushed away as scrap from the final finished object.

The strength and surface quality of the object depend on the shape of the object and, importantly, how the object is oriented as the object is built from sheet after sheet of laser-cut paper. In particular, if the density of the contour lines formed from the layers of sheets is too low, the scrap paper becomes difficult remove and this adversely affects the quality of the object's surface in such regions. No orientation will eliminate completely the quality problems due to regions of low contour line density because changing the orientation to improve one region is bound to shift the problem to another region of the object. However, the quality of certain regions such as screw threads or other regions that closely join with another part are more important to manufacture precisely.

Because the manufacturing process is slow, iterating the manufacturing process until a good quality orientation is found is not a tenable option. Moreover, picking good orientations is difficult for people to eye-ball because of the difficulty visualizing contour line density.

Bailey and Clark devised a texture-based interactive rendering technique using OpenGL for visualizing the contour line density of a 3D model. The idea is to color-code regions of contour line as a red-yellow-green spectrum with low density regions coded red (indicating poor contour line density and likely to lead to quality issues) and high density regions colored green (indicating good contour line density). With this color visualization, a manufacturing engineer can then interactively orient the model on a computer workstation based on the contour line density color-coding and knowledge of process-dependent design rules to find a suitable orientation for manufacturing the object.

The rendering technique uses 1D texturing and the texture matrix to render the model color-coded as described. Interactive changes to the model orientation require no more effort than updating the texture matrix and re-rendering the static geometry of the model. Because the technique leverages OpenGL's per-vertex transform and texturing functionality, graphics hardware that accelerates OpenGL's transform functionality and texturing will automatically accelerate this rendering technique.

Assume a coordinate space for surface normals with the Z axis perpendicular to the way the paper sheets are stacked. If the object's surface normals are transformed to this coordinate space, the contour line density is purely a function of the Z component of the surface normal. Specifically, the contour line density is:

$$\text{CLD} = \frac{\sqrt{1 - N_z}}{tN_z}$$

where t is the paper layer thickness, approximately 0.0042 inches.

The possible range of CLD can be color-coded as a red-yellow-green spectrum based on known design rules, and then the composite color-coded function can then be encoded in a 1D texture. In their particular case, Bailey and Clark found that a CLD below 100 causes problems for the manufacturing process. Therefore, the 1D texture would be set up to map a CLD below 100 to red.

Because t is a constant, CLD varies only with N_z. Typically, per-vertex surface normals are passed to OpenGL using `glNormal3f` calls and such normals are used for lighting. For this rendering technique however, the normalized per-vertex surface normal is passed to `glTexCoord3f` to serve as a 3D texture coordinate. Keep in mind that OpenGL transforms texture coordinates as 3D homogeneous values.

Then the texture matrix can be used to rotate this per-vertex surface normal (passed in to OpenGL as a 3D texture coordinate) to match the assumed coordinate space where the Z axis is perpendicular to the way the paper sheets are stacked. Because the normal N is assumed to be normalized, N_z will vary from $[-1, 1]$. Next, this rotated N_z component must be mapped to the s texture coordinate used for 1D texturing and scaled and biased to the $[0, 1]$ texture range. The rotation and scale and bias transformations are concatenated as shown:

$$\begin{pmatrix} s \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_1 & r_2 & r_3 & 0 \\ r_4 & r_5 & r_6 & 0 \\ r_7 & r_8 & r_9 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} N_x \\ N_y \\ N_z \\ 1 \end{pmatrix}$$

The composite of the rotate and scale and bias matrices can be loaded into the texture matrix like this:

```
GLfloat scaleBias[16] = { 0,   0, 0, 0,  /* OpenGL wants column major */
                          0,   0, 0, 0,
                          0.5, 0, 0, 0,
                          0.5, 0, 0, 1 };
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glMultMatrixf(&scaleBias[0]);
glRotatef(angle, x, y, z);
```

By enabling texturing and binding to the 1D color-coded texture and rendering the model using `glTexCoord3f` as described, the contour line density is effectively visualized. To re-render the model assuming a different orientation for the manufacturing process, simply adjust the texture matrix rotation and re-render the model's static geometry.

Note that there is no way to normalize a texture coordinate in the way that OpenGL supports `GL_NORMALIZE` for normalizing normals passed to `glNormal3f`. Also if rendering the model involves modelview matrix changes, these modelview matrix changes must also be reflected in the texture matrix.

The `NV_texgen_reflection_vector` [8] extension addresses both these issues. The extension provides two new texture coordinate generation (texgen) mode. The `GL_REFLECTION_MAP_NV` mode can place the eye-space

[8]The suffix *NV* indicates this extension was specified by NVIDIA Corporation. At the time of writing, `NV_texgen_reflection_vector` is implemented in NVIDIA's OpenGL and Mesa 3.1, an OpenGL-like API.

reflection vector coordinates in s, t, and r. The second mode is more interesting for the purpose at hand. The `GL_NORMAL_MAP_NV` mode can place the eye-space normal vector coordinates in s, t, and r. By using the `GL_NORMAL_MAP_NV` mode for the s, t, and r texture coordinates, you can use the technique described above, but simply call `glNormal3f` instead of `glTexCoord3f`. The `GL_NORMALIZE` functionality can be used to automatically normalize the per-vertex normals and modelview matrix changes are automatically accounted for.

6.16 Projective Textures

Projective textures [89] use texture coordinates which are computed as the result of a projection. The result is that the texture image can be subjected to a separate independent projection from the viewing projection. This technique may be used to simulate effects such as slide projector or spotlight illumination, to generate shadows, and to reproject a photograph of an object back onto the geometry of the object. Several of these techniques are described in more detail in later sections of these notes.

OpenGL generalizes the two component texture coordinate (s,t) to a four-component homogeneous texture coordinate (s,t,r,q). The q coordinate is analogous to the w component in the vertex coordinates. The r coordinate is used for three dimensional texturing in implementations that support that extension and is iterated in manner similar to $s$and t. OpenGL provides default values for r (0) and q (1). The addition of the q coordinate adds very little extra work to the usual texture mapping process. Rather than iterating (s,t,r) and dividing by 1/w at each pixel, the division becomes a division by q/w. Thus, in implementations that perform perspective correction there is no extra rasterization burden associated with processing q.

6.16.1 How to Project a Texture

Projecting a texture image into your synthetic environment requires many of the same steps that are used to project the rendered scene onto the display. The key to projecting a texture is the contents of the texture transform matrix. The matrix contains the concatenation of three transformations:

1. A modelview transform to orient the projection in the scene.
2. A projective transform (perspective or orthogonal).
3. A scale and bias to map the near clipping plane to texture coordinates.

The modelview and projection parts of the texture transform can be computed in the same way, with the same tools that are used for the modelview and projection transform. For example, you can use `gluLookat` to orient the projection, and `glFrustum` or `gluPerspective` to define a perspective transformation.

The modelview transform is used in the same way as it is in the OpenGL viewing pipeline, to move the viewer to the origin and the projection centered along the negative z axis. In this case, viewer can be thought of a light source, and the near clipping plane of the projection as the location of the texture image, which can be thought of as printed on a transparent film. Alternatively, you can conceptualize a viewer at the view location, looking through the texture on the near plane, at the surfaces to be textured.

The projection operation converts eye space into Normalized Device Coordinate (NDC) space. In this space, the x, y, and z coordinates range from -1 to 1. When used in the texture matrix, the coordinates are s, t, and r instead. The projected texture can be visualized as laying on the surface of the near plane of the oriented projection defined by the modelview and projection parts to the transform.

The final part of the transform scales and biases the texture map, which is defined in texture coordinates ranging from 0 to 1, so that the entire texture image (or the desired portion of the image) covers the near plane defined by the projection. Since the near plane is now defined in NDC coordinates, Mapping the NDC near plane to match the texture image would require scaling by $1/2$, then biasing by $1/2$, in both s and t. The texture image would be centered and cover the entire back plane. The texture could also be rotated if the orientation of the projected image needed to be changed.

The projections are ordered in the same as the graphics pipeline, the modelview transform happens first, then the projection, then the scale and bias to position the near plane onto the texture image:

1. `glMatrixMode(GL_TEXTURE)`
2. `glLoadIdentity( )` (clear current texture matrix)
3. `glTranslatef(.5f, .5f, 0.f)`
4. `glScalef(.5f, .5f, 1.f)` (texture covers entire NDC near plane)
5. Set the perspective transform (e.g., `glFrustum`).
6. Set the modelview transform (e.g., `gluLookAt`).

What about the texture coordinates for the primitives that the texture will be projected on? Since the projection and modelview parts of the matrix have been defined in terms of eye space (where the entire scene is assembled), the straightforward method is to create a 1-to-1 mapping between eye space and texture space. This can be done by enabling texture generation to eye linear and setting the eye planes to a one-to-one mapping:

- `GLfloat Splane[] = {1.f, 0.f, 0.f, 0.f};`
- `GLfloat Tplane[] = {0.f, 1.f, 0.f, 0.f};`
- `GLfloat Rplane[] = {0.f, 0.f, 1.f, 0.f};`
- `GLfloat Qplane[] = {0.f, 0.f, 0.f, 1.f};`

You could also use object space mapping, but then you'd have to take the current modelview transform into account.

So when you've done all this, what happens? As each primitive is rendered, texture coordinates matching the x, y, and z values that have been transformed by the modelview matrix are generated, then transformed by the texture transformation matrix. The matrix applies a modelview and projection transform; this orients and projects the primitive's texture coordinate values into NDC space (-1 to 1 in each dimension). These values are scaled and biased into texture coordinates. Then normal filtering and texture environment operations are performed using the texture image.

If transformation and texturing is being applied to all the rendered polygons, how do you limit the projected texture to a single area? There are a number of ways to do this. One is to simply only render the polygons you intend to project the texture on when you have projecting texture active and the projection in the texture transformation matrix. But this method is crude. Another way is to use the stencil buffer in a multipass algorithm to control what parts of the scene are updated by a projected texture. The scene can be rendered without the projected texture, the stencil buffer can be set to mask off an area, and the scene re-rendered with the projected texture, using the stencil buffer to mask off all but the desired area. This can allow you to create an arbitrary outline for the projected image, or to project a texture onto a surface that has a surface texture.

There is a very simple method that works when you want to project a non-repeating texture onto an untextured surface. Set the `GL_MODULATE` texture environment, set the texture repeat mode to `GL_CLAMP`, and set the texture border color to white. When the texture is projected, the surfaces outside the texture itself will default to the texture border color, and be modulated with white. This will leave the areas textured with the border color unchanged, since each color component will be scaled by one.

Filtering considerations are the same as for normal texturing; the size of the projected textures relative to screen pixels determines minification or magnification. If the projected image will be relatively small, mipmapping may be required to get good quality results. Using good filtering is especially important if the projected texture moves from frame to frame.

Please note that like the viewing projections, the texture projection is not really optical. Unless special steps are taken, the texture will affect all surfaces within the projection, both in front and in back of the projection. Since there is no implicit view volume clipping (like there is with the OpenGL viewing pipeline), the application needs to be carefully modeled to avoid undesired texture projections, or user defined clipping planes can be used to control where the projected texture appears.

6.17 Environment Mapping

OpenGL directly supports environment mapping using spherical environment maps. A sphere map is a single texture of a perfectly reflecting sphere in the environment where the viewer is infinitely far from the sphere. The environment behind the viewer (a hemisphere) is mapped to a circle in the center of the map. The hemisphere in front of the viewer is mapped to a ring surrounding the circle. Sphere maps can be generated using a camera with an extremely wide-angle (or fish eye) lens. Sphere map approximations can also be generated from a six-sided (or cube) environment map by using texture mapping to project the six cube faces onto a sphere.

OpenGL provides a texture generation function (`GL_SPHERE_MAP`) which maps a vertex normal to a point on the sphere map. Applications can use this capability to do simple reflection mapping (shade totally reflective surfaces) or use the framework to do more elaborate shading such as Phong lighting [95]. Applications of environment mapping are discussed in Sections 10.4 and 11.2.1.

6.18 Image Warping and Dewarping

Image warping or dewarping may be implemented using texture mapping by defining a correspondence between a uniform polygonal mesh and a warped mesh. The points of the warped mesh are assigned the corresponding texture coordinates of the uniform mesh and the mesh is texture mapped with the original image. Using this technique, simple transformations such as zoom, rotation or shearing can be efficiently implemented. The technique also easily extends to much higher order warps such as those needed to correct distortion in satellite imagery.

6.19 3D Textures

Three dimensional textures are a logical extension of 2D textures. In 3D textures, texels become unit cubes in texel space. They are packed into a rectangular parallelepiped, each dimension constrained to be a power of two. This texture map occupies a volume, rather than a rectangular region, and is accessed using three texture coordinates; s, t, and r. As with 2D textures, texture coordinates range from 0 to 1 in each dimension. Filtering is controlled in the same fashion as 2D textures, using texture parameters and texture environment.

6.19.1 Using 3D Textures

In OpenGL, 3D textures have much in common with 2D and 1D textures. Texture parameters and texture environment calls are the same, using the `GL_TEXTURE_3D_EXT` (`GL_TEXTURE_3D` in OpenGL 1.2) target in place of `GL_TEXTURE_2D` or `GL_TEXTURE_1D`.

Internal and external formats and types are the same, although a particular OpenGL implementation may limit the availability of 3D texture formats.

3D textures need to be accessed with s, t, and r texture coordinates instead of just s and t. The additional texture coordinate complexity, combined with the common uses for 3D textures, means texture coordinate generation is used more commonly for 3D textures than for 2D and 1D.

3D texture maps take up a large amount of texture memory, and are expensive to change dynamically. This can affect multipass algorithms that require multiple passes with different textures.

The texture matrix operates on 3D texture coordinates in the same way that it does for 2D and 1D textures. A 3D texture volume can be translated, rotated, scaled, or have other transforms applied to it. Applying a transformation to the texture matrix is a convenient and high performance way to manipulate a 3D texture when it is too expensive to alter the texel values directly.

3D Textures vs. Mipmaps A clear distinction should be made between 3D textures and mipmapped 2D textures. 3D textures can be thought of as a solid block of texture, requiring a third texture coordinate r, to access any given texel. A 2D mipmap is a series of 2D texture maps, each filtered to a different resolution. Texels from the appropriate level(s) are chosen and filtered, based on the relationship between texel and pixel size on the primitive being textured.

Like 2D textures, 3D texture maps can be mipmapped. Instead of resampling a 2D layer, the entire texture volume is filtered down to an eighth of its volume by averaging eight adjacent texels on one level down to a single texel on the next. Mipmapping serves the same purpose in both 2D and 3D texture maps; it provides a means of accurately filtering when the projected texel size is small relative to the pixels being rendered.

6.19.2 3D Textures to Render Solid Materials

A direct 3D texture application is rendering solid objects composed of heterogeneous material. An example is rendering a statue made of marble or wood. The object itself is composed of polygons or NURBS surfaces bounding the solid. Combined with proper texgen values, rendering the surface using a 3D texture of the material makes the object appear cut out of the material. With 2D textures objects often appear to have the material laminated on the surface. The difference can be striking when there are obvious 3D coherencies in the material, combined with sharp angles in the object's surface.

Rendering a solid with 3D texture is straightforward:

Create the 3D texture The texture data for the material is organized as a three dimensional array. Often the material is generated procedurally. As with 2D textures, proper filtering and sampling of the data must be done to avoid aliasing. A mipmapped 3D texture will increase realism of the object. OpenGL does not support a `gluBuild3DMipmaps` command, so the mipmaps need to created by the application. Be sure to check to see if the size of the texture you want to create is supported by the system, and there is sufficient texture memory available by calling `glTexImage3DEXT` with `GL_PROXY_TEXTURE_3D_EXT` to find a supported size. You can also call `glGet` with `GL_MAX_3D_TEXTURE_SIZE_EXT` to find the maximum allowed size of any dimension in a 3D texture for your implementation of OpenGL, though the result may be more conservative than the result of a proxy query.

Create Texture Coordinates For a solid surface, using `glTexGen` to create the texture coordinates is the easiest approach. Define planes for s, t, and r in eye space. Adjusting the scale has more effect on texture quality than the position and orientation of the planes, since scaling affects how the texture is sampled.

Enable Texturing Use `glEnable(GL_TEXTURE_3D_EXT)` to enable 3D texture mapping. Be sure to set the texture parameters and texture environment appropriately. Check to see what restrictions your implementation puts on these values.

Render the Object Once configured, rendering with 3D texture is no different than other texturing.

6.19.3 3D Textures as Multidimensional Functions

Instead of thinking of a 3D texture as a 3D volume of data, it can be thought of as a 2D texture map that varies as a function of the r coordinate value. Since the 3D texture filters in three dimensions, changing the r value smoothly

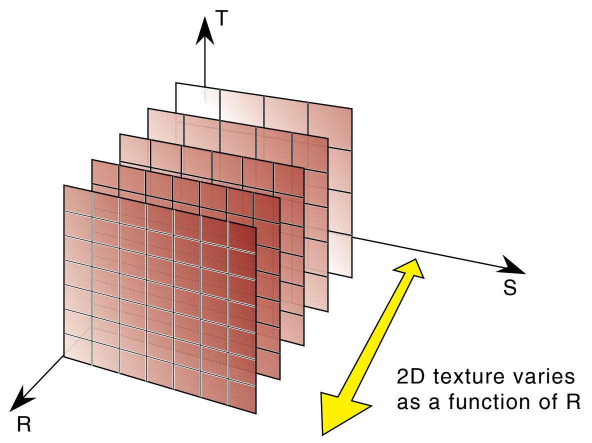


Figure 39. 3D Textures as 2D Textures Varying with R

blends from one 2D texture image to the next.

An obvious application is animated 2D textures. A 3D texture can animate a sequence of images by using the r value as time. Since the images are interpolated, temporal aliasing is reduced.

Another application is generalized billboards. A normal billboard is a 2D texture applied to a polygon that always faces the viewer. Billboards of objects such as trees behave poorly when the viewer views the object from above. A 3D texture billboard can change the textured image as a function of viewer elevation angle, blending a sequence of images between side view and top view, depending on the viewer's position.

6.20 Detail Textures

Texture filtering can become unrealistic when magnifying. When the viewer is close to a texture surface, and single texels start to cover many pixels. The linear magnification filtering of these texels results in an unrealistically smoothed image with little surface detail. Not only does the image look unrealistic, but the lack of high frequency spatial information on the surface makes it more difficult to get realistic height and and motion cues when moving over the surface.

Ideally, every texture will have enough fine levels that any normal view of the textured surface will always have sufficient high frequency spatial data. But providing extra levels are expensive. With mipmapping, each fine level requires four times as many texels as the next coarser one. In some cases, it is worth it. The finer levels contain much more visual information that's useful to the application.

But sometimes it is not. A very high resolution image of an object will contain surface details, but the details can be very similar across the surface. For example, a close-up photo of a road may show a lot of asphalt detail that's pretty similar across the entire road. Providing a mipmap level of this detail would consume a lot of texture memory, without adding a lot of useful image data. Yet this detail provides important motion and height cues, and keeps the level from looking too blurry.

A detail texture is one solution to this problem. A representative section of a high resolution image is chosen, and its high frequency information extracted. The extracted information is stored in a small texture that contains just a fraction of the entire image.

The main mipmapped textured can then have fewer, lower resolution levels. When the viewer is close to the textured surface, the detail texture is combined with the filtered base texture to provide high frequency information

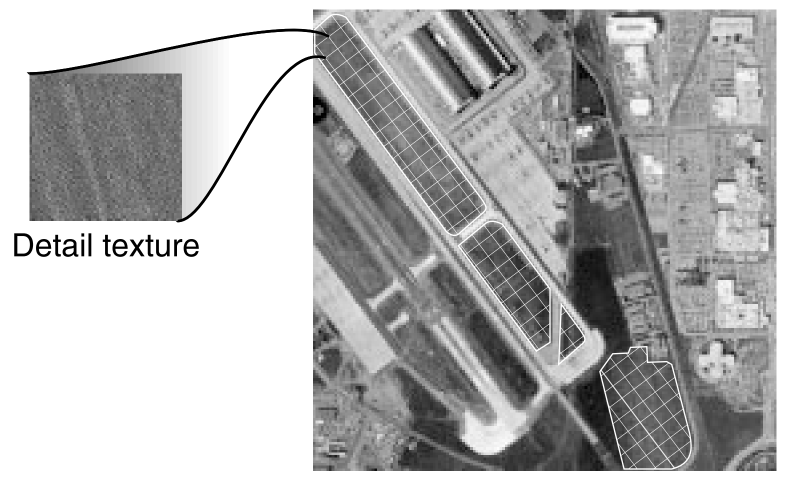


Figure 40. Detail Textures

to the result. Since the detail texture is small, its pattern is repeated over the entire visible surface.

It is assumed that the detailed texture contains only high frequency image features. These features are changing rapidly even across a small detail texture, so there are no low frequency components to cause tiling artifacts when repeating the detail texture across the textured surface.

Detail textures should not contribute anything to a texture that is not magnifying. When implementing detail texturing, you must be careful to fade in detail texturing as a function of the magnification of the base texture.

One way to do this is to gradually blend in the detail texture contribution as a function of distance from the textured surface. In many cases, application specific constraints can simplify the problem. For example, a flight simulator may have a look down mode that only needs a height above ground and a precomputed scaling factor to determine magnification level. If the simulator's view frustum brings the entire visible textured surface into view at nearly the same magnification, this solution can work well.

In the general case however, computing texture magnification can be difficult. You must consider the visible vertices of the textured surface, the texture coordinate scaling resulting from the current modelview and projection transformations, the current texture generations settings, and the values in the texture transformation matrix. One way around this is to add detail texture support in the OpenGL implementation. This is done in the detail texture extension `GL_SGIS_detail_texture` supported on SGI hardware. This extension blends in the detail texture as a function of magnification, and allows the detail texture either to add to or modulate the base texture.

6.20.1 Signed Intensity Detail Textures

One technique that avoids having to compute the base texture magnification is to create a signed detail texture. The detail texture image created so that it has both positive and negative intensity values, with an average value over the detail texture of zero; when combined with the base image, it modifies it, adding high frequency components to the textured image. The detail texture is combined with the base texture in a separate pass, using alpha blending.

Different blend functions can be used, depending on whether you want to add in the detail texture or modulate with it. In the first pass the image is drawn with the base texture, in the second pass, The detail texture is made current, and since it is higher resolution, the texture coordinate mapping is changed, either by changing the texgen mapping or with the texture transformation matrix. Blending mode is enabled, and the blend function is set. If the

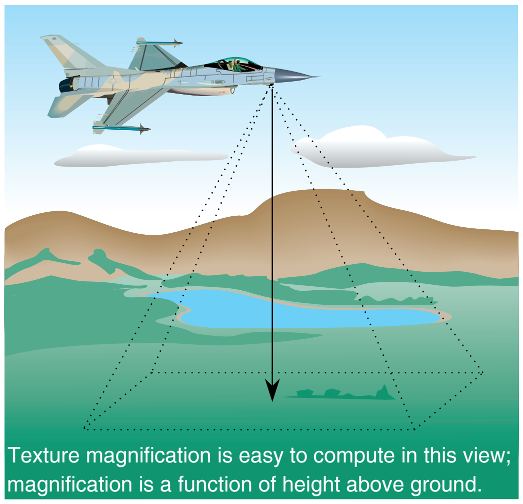

Texture magnification is easy to compute in this view;
magnification is a function of height above ground.

Figure 41. Special Case Texture Magnification

blend function is `glBlendFunc(GL_ONE, GL_ONE)`, the detail texture is added to the base texture. If the blend function is `glBlendFunc(GL_ZERO, GL_SRC_COLOR)`, the detail texture will modulate the base texture.

The clever part of this algorithm is how the detail texture combines with the base texture as a function of magnification. The detail texture is applied to the same geometry as the base texture. The texturing system is configured so that the detail texture is at an offset magnification value relative to the base texture; it minifies if the base texture is not magnifying. The minification filtering will cause the signed intensity components to blend together.

If the average intensity of the detail texture is zero, it will have little or no contribution to the image. As both the detail and base texture are zoomed, the filtering of the detail texture begins to magnify, and the signed intensity values stop canceling each other out.

Although a signed texture value can not be blended directly, it can be simulated by using a subtractive blend and a biasing term. The signed texels of the detail texture are first converted to positive values. For example, if the texture values range from -1/4 to 1/2, the texels can be biased by 1/4. Then the texture images applied and blended normally. After the two textures are combined, a third pass subtracts out the 1/4 bias term from the textured image.

1. Create a signed detail texture image ranging from -1/4 to 1/2.
2. Bias the image to make it non-negative.
3. Render the surface with the base texture.
4. Enable blending.
5. Set blend function to modulate or add.
6. Re-render the surface using the detail texture with different texture coordinates.
7. `glBlendEquation(GL_FUNC_REVERSE_SUBTRACT)`
8. Render the image unlit with a gray color (equal to the bias term) to remove the biasing term.

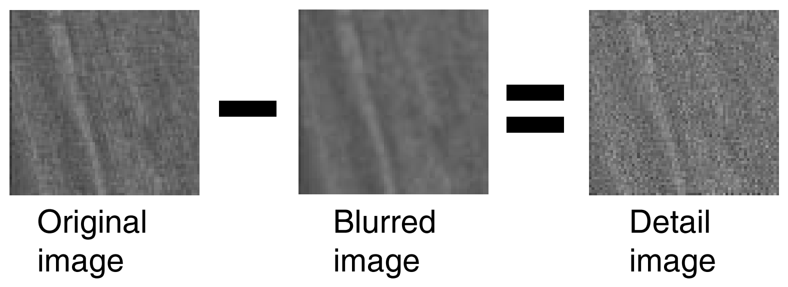


Figure 42. Subtracting out Low Frequencies

6.20.2 Making Detail Textures

Detail textures contain the high frequency components from the texture image. The high frequency information is extracted, not generated from scratch. So you must start with a high resolution version of the desired texture.

The first step is to choose the size of the detail texture, and select a region of the detailed image that contains high frequency details representative of the entire image. Now extract the high frequency components of that region. One technique is to remove the high frequency components from one copy of the region by blurring it. This can be done with an image processing application, or you can use `gluScaleImage` to scale the image down, then up again. For more sophisticated filtering, you can use a blurring convolution kernel, assuming your implementation of OpenGL supports the imaging subset. Enable convolution, set the appropriate blurring filter kernel and use `glCopyPixels` to process the image.

Now subtract the blurred image from the unprocessed one. You can do this using the subtractive blend mode or with the accumulation buffer. The result will be a signed image that contains the high frequency components of the image. You will have to be careful to add a biasing value before subtracting (or before returning the image from the accumulation buffer) to avoid negative pixel values, since the frame buffer will clamp them. If you have the imaging subset, you can use the minmax feature to find the range of pixel values in both the sharp and blurry parts of the detail texture image before you subtract them. You can then use the results to find the proper biasing term.

6.21 Procedural Texture Generation

Procedurally generated textures are a diverse topic; we concentrate on those based on *filtered noise functions* . They are commonly used to simulate effects from phenomena such as fire, smoke, clouds, and marble formation. These textures are described in detail in [25], which provides the basis for much of this section.

6.21.1 Filtered Noise Functions

A filtered noise function is simply a function created by filtering impulses of random amplitude over the domain. There are a variety of ways to distribute the impulses spatially and to filter those impulses; these methods determine the character of the function and, in turn, the character of the procedural texture created from the function. Regardless of the method chosen, a filtered noise function should have certain properties [25], some of which are:

- It is a repeatable pseudorandom function of its inputs.
- It has a known range, typically -1 to 1.

- It is band-limited, with a maximum frequency of about 1 per domain unit.

Given such a function, we can build a more interesting function by making dilated versions of the original such that each one has a frequency of 2, 4, 8, etc. These are called the *octaves* of the original function. The octaves are then composited together with the original noise function using some set of weights. The result is a band-limited function which gives the impression of controlled randomness in each frequency band.

One way of distributing noise impulses is to space them uniformly along the coordinate axes, as in a lattice. In *value noise,* the function itself interpolates the values at the lattice points, while in *gradient noise* the gradient of the function interpolates the values at the lattice points [25]. Gradient noise is similar to the noise function implemented in the RenderMan shading language.

Lattice noises can exhibit axis-aligned artifacts. Lewis [58] describes *sparse convolution,* a way to avoid such artifacts by distributing the impulses using a stochastic process, and van Wijk [97] describes a similar technique called *spot noise*.

Although the noise functions described in [25] are generally 3D, we first discuss how to generate a 2D noise function, because it is more straightforward to construct in a 2D framebuffer and because some simple interesting effects can be created with it.

6.21.2 Generating Noise Functions

Filtered noise functions are typically implemented as continuous functions that can be sampled at an arbitrary domain value. However, for some applications a set of uniformly spaced samples of the function may suffice. In these cases, a discrete version of the function can be created in the framebuffer using OpenGL. In the following, we do not distinguish between the terms *noise function* and *discrete noise function* .

A simple way to create lattice noise is to create a texture with random values for the texels, and then to draw a textured rectangle with a bilinear texture filter at an appropriate magnification. However, bilinear interpolation produces poor results, especially when creating the lower octaves, where values are interpolated across a large area. Some OpenGL implementations support bicubic texture filtering, which may produce results of acceptable quality. However, a particular implementation of bicubic filtering may have limited subtexel precision, causing noticeable banding at the lower octaves. Both bilinear and bicubic filters also have the limitation that they produce only value noise; gradient noise is not possible. We suggest another approach.

6.21.3 High Resolution Filtering

The accumulation buffer can be used to convolve a high resolution filter with a relatively small image under magnification. That is what we need to make the different octaves; the octave representing the lowest frequency band will be created from a very small input image under large magnification. Suppose we want to create a 512x512 output image by convolving a 64x64 filter with a 4x4 input image. Our filter takes a 2x2 array of samples from the input image at a time, but is discretized into 64x64 values in order to generate an output image of the desired size. The input image is shown on the left in Figure 43 with each texel numbered. The output image is shown on the left in Figure 44. Note that each texel of the input image will make a contribution to a 64x64 region of the output image. Consider these regions for texels 5, 7, 13, and 15 of the input image; they are adjacent to each other and have no overlap, as shown by the dotted lines on the left in Figure 44. Hence, these four texels can be evaluated in the same pass without interfering with each other. Making use of this fact, we redistribute the texels of the input image into four 2x2 textures as shown in the right of Figure 43. We also create a 64x64 texture that contains the filter function; this texture will be used to modulate the contribution of the input texel over a 64x64 region of the color buffer. The steps to evaluate the texels in Texture D are:

1. Using the filter texture, draw four filter functions into the alpha planes with the appropriate x and y offset, as shown on the right in Figure 44.

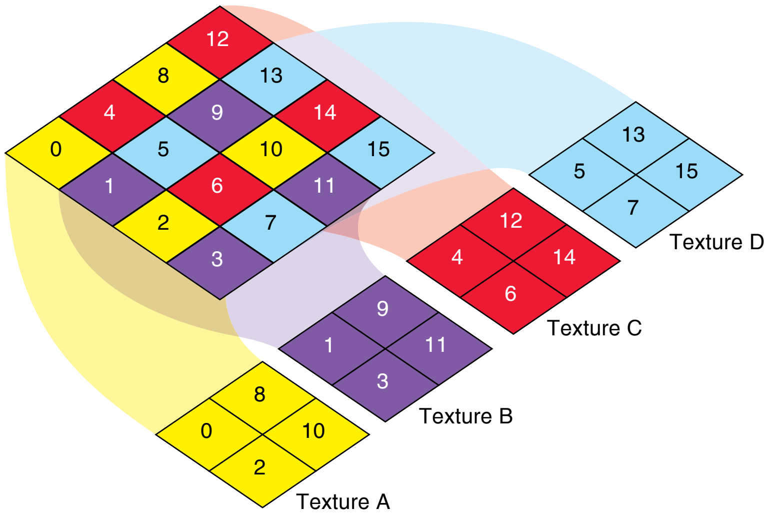


Figure 43. Input Image

2. Enable alpha blending and set the source blend factor to `GL_DST_ALPHA` and the destination blend factor to `GL_ZERO`.
3. Set the texture magnification filter to `GL_NEAREST`.
4. Draw a rectangle to the dotted region with Texture D, noting the offset of 64 pixels in both x and y.
5. Accumulate the result into the accumulation buffer.

Repeat the above procedure for Textures A, B, and C with the appropriate x and y offsets, and return the contents of the accumulation buffer to the color buffer.

A wider filter requires more passes of the above procedure, and also requires that the original texture be divided into more small textures. For example, if we had chosen a filter that covers a 4x4 array of input samples instead of 2x2, we would have to make 16 passes instead of 4, and we would have to distribute the texels into 16 1x1 textures. Increasing the size of either the output image or the input image, however, has no effect on the number of passes.

6.21.4 Spectral Synthesis

Now that we can create a single frequency noise function using the framebuffer, we need to create the different octaves and to composite them into one texture. For each octave:

1. Scale the texture matrix by a power of 2 in both s and t.
2. Translate the texture matrix by a random offset in both s and t.
3. Set the texture wrap mode to `GL_REPEAT` for s and t.
4. Draw a textured rectangle.
5. Accumulate the color buffer contents.

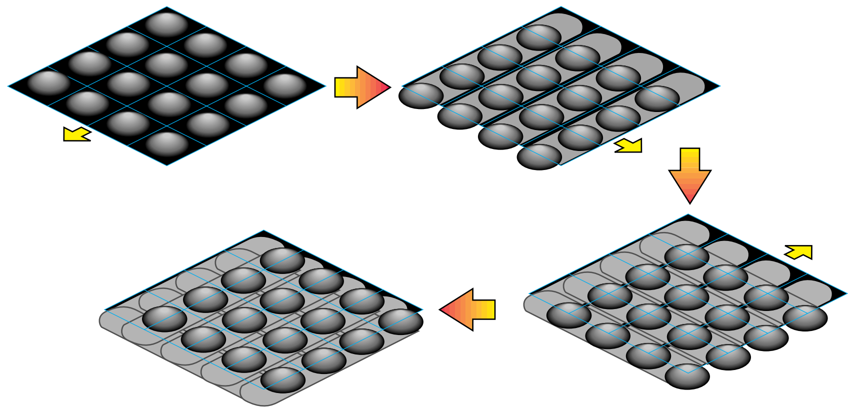


Figure 44. Output Image

The random translation is an attempt to minimize the amount of overlap between each octave's texels; without it, every octave would use texels from the same corner of the input image. The accumulation is typically done with a scale factor that controls the weight we want to give each octave.

6.21.5 Other Noise Functions

Gradient noise can be created using the same method described above, but with a different filter. The technique described above can also create noise that is not aligned on a lattice. To create sparse convolution noise [58] or spot noise [97], instead of drawing the entire point-sampled texture at once, draw one texel and one copy of the filter at a time for each random location.

6.21.6 Turbulence

To create an illusion of turbulent flow, first-derivative discontinuities are introduced into the noise function by taking the absolute value of the function. Although OpenGL does not include an absolute value operator for framebuffer contents, the same effect can be achieved by the following:

1. `glAccum(GL_LOAD,1.0);`
2. `glAccum(GL_ADD,-0.5);`
3. `glAccum(GL_MULT,2.0);`
4. `glAccum(GL_RETURN,1.0);`
5. Save the image in the color buffer to a texture, main memory, or other color buffer.
6. `glAccum(GL_RETURN,-1.0);`
7. Draw the saved image from Step 5 using `GL_ONE` as both the source blend factor and the destination blend factor.

The calls with `GL_ADD` and `GL_MULT` map the values in the accumulation buffer from the range [0,1] to [-1,1]; this is needed because values retrieved from the color buffer into the accumulation buffer are positive. Since values from the accumulation buffer are clamped to [0,1] when returned, the first `GL_RETURN` clamps all negative values to 0 and returns the positive values intact. The second `GL_RETURN` clamps the positive values to 0, and negates and returns the negative values. The color buffer needs to be saved after the first `GL_RETURN` because the second `GL_RETURN` overwrites the color buffer; OpenGL does not define blending for accumulation buffer operations.

6.21.7 Example: Image Warping

A common use of a 2D noise texture is to distort the texture coordinates while drawing a 2D image, thus warping the image. A noise function is created in the framebuffer as described above, read back to the host, and used as texture coordinates (or offsets to texture coordinates) to render the image. Since color values in OpenGL are normalized to the range 0.0 to 1.0, if one is careful the image returned to the host may be used without much conversion; assuming that the modelview and texture matrixes are set up to accept values in this range, the returned data may be used directly for rendering.

Another similar use of a 2D noise texture is to distort the reflection of an image. In OpenGL, reflections on a flat surface can be done by reflecting a scene across the surface. The results can be copied from the framebuffer to texture memory, and in turn drawn with distorted texture coordinates. The shape and form of the distortion can be controlled by modulating the contents of the framebuffer after the noise texture is drawn but before it is copied to texture memory. This can produce interesting effects such as water ripples.

6.21.8 Generating 3D Noise

Using the techniques described above for generating a 2D noise function, we can generating a 3D noise function by making 2D slices and filtering them. A 2D slice spans the s and t axes of the lattice, and corresponds to a slice of the lattice at a fixed r.

Suppose we want to make a 64x64x64 noise function with a frequency of 1 per domain unit, using the same filtering (but one that now takes 2x2x2 input samples) as in the 2D example above. We first create 2 slices, one for r= 0.0 and one for r =1.0. Then we create the 62 slices in between 0 and 1 by interpolating the two slices. This interpolation can take place in the color buffer using blending, or it can take place in the accumulation buffer. Functions with higher frequencies are created in a similar way. Widening the filter dramatically increases the number of passes; going from a 2x2x2 filter to 4x4x4 requires 16 times as many passes.

To synthesize a function with different frequencies, we create a 3D noise function for each frequency, and composite the different frequencies using a set of weights, just as we do in the 2D case. It is clear that a large amount of memory is required to store the different 3D noise functions. These operations may be reordered so that less total memory is required, perhaps at the expense of more interpolation passes.

6.21.9 Generating 2D Noise to Simulate 3D Noise

We have described a method for creating 2D noise functions. In the case of lattice noise, these 2D functions correspond to a 2D slice of the lattice. There are cases where we want to model a 3D noise function and where such a 2D function is inadequate. For example, to draw a vase that looks like it was carved from a solid block of marble, we cannot use a lattice 2D noise function.

However, we can create a 2D noise function that approximates the appearance of a true 3D noise function, using spot noise [97]. We take into account the object space coordinates of the geometry, and generate only spots that are close enough to the geometry to make a contribution to the 3D noise at those points. The difficulty is how to render the spot in such a way that at each fragment the value of the spot is determined by the object space distance from the center of the spot to that fragment. Depending on the complexity of the geometry, we may be able to make an acceptable approximation to the correct spot value by distorting the spot texture. One possible way to improve the approximation is to compensate for a nonuniform mapping of the noise texture to the geometry. Van Wijk describes how he does this by nonuniformly scaling a spot. Approximating the correct spot value is most important when generating the lower octaves, where the spots are largest and errors are most noticeable.

6.21.10 Trade-offs Between 3D and 2D Techniques

A 3D texture can be used with arbitrary geometry without much additional work if your OpenGL implementation supports 3D textures. However, generating a 3D noise texture requires a large amount of memory and a large number of passes, especially if you use a filter that convolves a large number of input values at a time. A 2D texture as we just described doesn't require nearly as many passes to create, but it does require knowledge of the geometry and additional computation in order to properly shape the spot.

7 Line Rendering Techniques

7.1 Wireframe Models

If your goal is to draw a true wireframe model, as opposed to drawing a hidden line rendering of a model or highlighting edges of a model, there are several methods available (listed here in order of least efficient to most efficient):

1. Draw the model as polygons in line mode using `glBegin(GL_POLYGON)` and `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`.

   This method is by far the easiest if you're already displaying the model as a shaded solid, since it involves a single mode change. However, it is likely to be significantly slower than the other methods both because more processing usually occurs for polygons than for lines and because every edge that is common to two polygons will be drawn twice. This method is undesirable when using antialiased lines as well, because each line that is drawn twice will be brighter than any lines drawn just once.

2. Draw the polygons as line loops using `glBegin(GL_LINE_LOOP)`.

   This method is almost as simple as the first, requiring only a change to the `glBegin` call. However, except for possibly eliminating the extra processing required for polygons it has all of the other undesirable features as well.

3. Extract the edges from the model and draw as independent lines using `glBegin(GL_LINES)`.

   This method is more work than the previous two because each edge must be identified and all duplicates removed. However, the extra work must only be done once and every time the model is drawn it will be drawn much faster.

4. Extract the edges from the model and connect as many as possible into long line strips using `glBegin(GL_LINE_STRIP)`.

   For just a little bit more effort than the `GL_LINES` method, lines sharing common end-points can be connected into larger line strips. This has the advantage of requiring less storage, less data transfer bandwidth, and makes most efficient use of any line drawing hardware.

7.2 Hidden Lines

This section describes techniques to draw wireframe objects with the hidden lines removed or drawn in a style different from the ones that are visible. This technique can clarify complex line drawings of objects, and improve their appearance [52] [5].

The algorithm assumes that the object is composed of polygons. The algorithm first renders the polygons of the objects, then the edges themselves, which make up the line drawing. During the first pass, only the depth buffer is updated. During the second pass, the depth buffer only allows edges that are not obscured by the object's polygons to be rendered, leaving the previous contents of the frame buffer undisturbed everywhere an edge is not drawn.

Here's the algorithm in detail:

1. Disable writing to the color buffer with `glColorMask`.
2. Enable depth testing with `glEnable(GL_DEPTH_TEST)`.
3. Render the object as polygons.
4. Enable writing to the color buffer.

5. Render the object as edges using one of the methods described in Section 7.1.

For best results the lines should be offset from the polygons using either `glPolygonOffset` or `glDepthRange` to help reduce depth buffer aliasing artifacts.

The stencil buffer may be used to avoid the depth buffering artifacts for convex objects drawn using non-antialiased (jaggy) lines all of one color. The following technique uses the stencil buffer to mask where all the lines are (both hidden and visible). Then it uses the stencil function to prevent the polygon rendering from updating the depth buffer where the stencil values have been set. When the visible lines are rendered, there is no depth value conflict, since the polygons never touched those pixels.

Here's the modified algorithm:

1. Disable writing to the color buffer with `glColorMask`.
2. Disable depth testing; `glDisable(GL_DEPTH_TEST)`.
3. Enable stenciling; `glEnable(GL_STENCIL_TEST)`.
4. Clear the stencil buffer.
5. Set the stencil buffer to set the stencil values to 1 where pixels are drawn; `glStencilFunc(GL_ALWAYS, 1, 1)`; `glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE)`.
6. Render the object as edges.
7. Use the stencil buffer to mask out pixels where the stencil value is 1; `glStencilFunc(GL_EQUAL, 1, 1)` and `glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP)`.
8. Render the object as polygons.
9. Turn off stenciling `glDisable(GL_STENCIL_TEST)`.
10. Enable writing to the color buffer.
11. Render the object as edges using one of the methods described in Section 7.1.

Variants of the above algorithm may be applied to each convex part of an object or, if the topology of the object is not known, to each individual polygon to render well-behaved hidden line images.

Instead of removing hidden lines, sometimes it's desirable to render them with a different color or pattern. This can be done with a modification of the algorithm:

1. Leave the color depth buffer enabled for writing.
2. Set the color and/or pattern you want for the hidden lines.
3. Render the object as edges.
4. Disable writing to the color buffer.
5. Render the object as polygons.
6. Set the color and/or pattern you want for the visible lines.
7. Render the object as edges using one of the methods described in Section 7.1.

In this technique, all the edges are drawn twice; first with the hidden line pattern, then with the visible one. Rendering the object as polygons updates the depth buffer, preventing the second pass of line drawing from effecting the hidden lines.

7.2.1 glPolygonOffset

In addition to the above methods which enable and disable various modes during the two passes of rendering, the `glPolygonOffset` command may be used to move the lines and polygons relative to each other. If the edges are drawn as lines in polygon mode, `glEnable(GL_POLYGON_OFFSET_LINE)` can be used to move the lines a little bit in front of the polygons. If a faster version of drawing the lines is used (as described in Section 7.1), `glEnable(GL_POLYGON_OFFSET_FILL)` will move the polygon surfaces a little bit behind the lines.

Keep in mind, however, that `glPolygonOffset` is designed to provide greater offsets for polygons viewed more edge-on than for polygons that are flatter relative to the screen. This means that additional work is done for each polygon which could slow down rendering. An advantage, however, is that once the parameters have been tuned for a particular OpenGL implementation, the same unmodified code should work well on other implementations.

7.2.2 glDepthRange

Similar effects are available using `glDepthRange` but both the polygons and the edges are drawn at the maximum speed for each type of primitive. This is done by moving the *zNear* value out a little bit from 0.0 while setting the *zFar* to 1.0 for all normal drawing. Then when the edges are drawn move the *zNear* value to 0.0 and reduce the *zFar* value by the same amount. The offset should be *at least* 0.00001, depending on the depth buffer accuracy and the amount of perspective used in the projection matrix, and may need to be significantly greater in many cases.

The general algorithm for an offset of EDGE_OFFSET is:

```
glDepthRange(EDGE_OFFSET, 1.0);
<draw all non-edge geometry>

glDepthRange(0.0, 1.0 - EDGE_OFFSET);
<draw all edges>
```

As with all algorithms described in this manual, it is up to the user to select the hidden line (or edge highlighting) method that best meets his needs after considering ease of implementation, speed, and image quality.

7.3 Haloed Lines

Haloing lines can make it easier to understand a wireframe drawing. Lines that pass behind other lines stop short a little before passing behind. It makes it clearer which line is in front of the other.

Haloed lines can be drawn using the depth buffer. The technique has two passes. First disable writing to the color buffer; the first pass only updates the depth buffer. Set the line width to be greater than the normal line width you're using. The width you choose will determine the extent of the halos. Render the lines. Now set the line width back to normal, and enable writing to the color buffer. Render the lines again. Each line will be bordered on both sides by a wider "invisible line" in the depth buffer. This wider line will mask out other lines as they pass beneath it.

1. Disable writing to the color buffer.
2. Enable the depth buffer for writing.
3. Increase line width.
4. Render lines.
5. Restore line width.

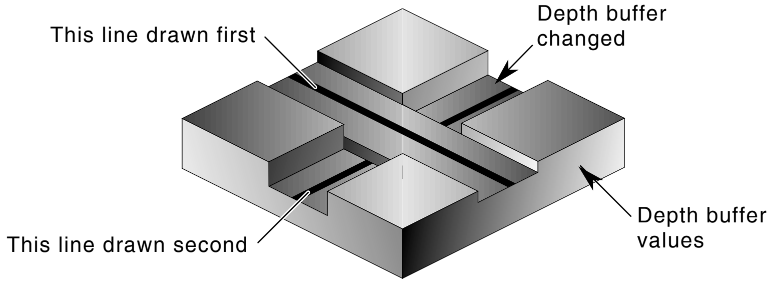


Figure 45. Haloed Line

6. Enable writing to the color buffer.
7. Ensure that depth testing is on, passing on `GL_LEQUAL`.
8. Render lines.

This method will not work where multiple lines with the same depth meet. Instead of connecting, all of the lines will be “blocked” by the last wide line drawn. There can also be depth buffer aliasing problems when the wide line z values are changed by another wide line crossing it. This effect becomes more pronounced if the narrow lines are widened to improve image clarity.

To avoid this problem, use polygon offset to move narrower visible lines in front of the obscuring lines when the lines are being drawn as polygons in line mode. The minimum offset should be used to avoid lines from one surface of the object “popping through” the lines of a another surface separated by only a small depth value.

If the vertices of the object’s faces are oriented to allow face culling, then face culling can be used to sort the object surfaces and allow a more robust technique: the lines of the object’s back faces are drawn, then obscuring wide lines of the front face are drawn, then finally the narrow lines of the front face are drawn. No special depth buffer techniques are needed.

1. Cull the front faces of the object.
2. Draw the object as lines.
3. Cull the back faces of the object.
4. Draw the object as wide lines in the background color.
5. Draw the object as lines.

Since the depth buffer isn’t needed, there are no depth aliasing problems. The backface culling technique is fast and works well, but is not general. It won’t work for multiple obscuring or intersecting objects.

7.4 Silhouette Edges

Sometimes it can be useful for highlighting purposes to draw a silhouette edge around a complex object. A silhouette edge defines the outer boundaries of the object with respect to the viewer as shown in Figure 46.

The stencil buffer can be used to render a silhouette edge around an object. With this technique, you can render the object, then draw a silhouette around it, or just draw the silhouette itself [84].

The object is drawn 4 times; each time displaced by one pixel in the x or y direction. This offset must be done in window coordinates. An easy way to do this is to change the viewport coordinates each time, changing the viewport transform. The color and depth values are turned off, so only the stencil buffer is affected.

Every time the object covers a pixel, it increments the pixel's stencil value. When the four passes have been completed, the perimeter pixels of the object will have stencil values of 2 or 3. The interior will have values of 4, and all pixels surrounding the object exterior will have values of 0 or 1.

Here is the algorithm in detail:

1. If you want to see the object itself, render it in the usual way.
2. Clear the stencil buffer to zero.
3. Disable writing to the color and depth buffers.
4. Set the stencil function to always pass, set the stencil operation to increment.
5. Translate the object by +1 pixel in y, using `glViewport`.
6. Render the object.
7. Translate the object by -2 pixels in y, using `glViewport`.
8. Render the object.
9. Translate by +1 pixel x and +1 pixel in y.
10. Render.
11. Translate by -2 pixel in x.
12. Render.
13. Translate by +1 pixel in x. You should be back to the original position.
14. Turn on the color and depth buffer.
15. Set the stencil function to pass if the stencil value is 2 or 3. Since the possible values range from 0 to 4, the stencil function can pass if stencil bit 1 is set (counting from 0).
16. Rendering any primitive that covers the object will draw only the pixels of the silhouette. For a solid color silhouette, render a polygon of the color desired over the object.

One of the bigger drawbacks of this algorithm is that it takes a large number of drawing passes to generate the edges. A somewhat more efficient algorithm suggested by Akeley[4] is to use `glPolygonOffset` to draw an offset depth image and then draw the polygons using `GL_LINE` polygon mode. The stencil buffer is again used to count the number of times each pixel is written. However, instead of counting the absolute number of writes to a pixel, the stencil value is inverted on each write. The resulting stencil buffer will contain a one wherever a pixel has been drawn an odd number of times. This ensures that lines drawn at the shared edges of polygon faces have stencil values of zero since the lines will be drawn twice. While this algorithm is a little more approximate then the previous algorithm it only requires two passes through the geometry.

The faster algorithm does not generate quite the same result as the first algorithm since it counts even and odd transitions and relies on the depth image to ensure that other non-visible surfaces do not interfere with the stencil count. The differences arise in that boundary edges within one object that are in front of another object will

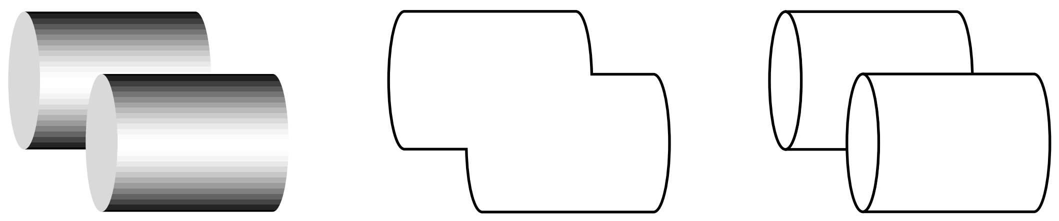


Figure 46. Shaded Solid Image, Silhouette Edges, Silhouette and Boundary Edges

be rendered as part of the silhouette image. By boundary edges we mean the *true* edges edges of the modeled geometry and do not include the interior shared-face edges. In many cases this artifact is useful as silhouette edges by themselves often do not provide sufficient information about the shape of objects. It is possible to combine the algorithm for drawing silhouettes with an additional step in which all of the boundary edges of the geometry are drawn as lines to produce a hidden line drawing displaying boundary edges plus silhouette edges.

The steps of the combined algorithm are:

1. Clear the depth and color buffers and clear the stencil buffer to zero.
2. Disable color buffer writes.
3. Draw the depth buffered geometry using `glPolygonOffset` to offset the image towards the far clipping plane.
4. Disable writing to the depth buffer and `glPolygonOffset`.
5. Set the stencil function to always pass and set the stencil operation to invert.
6. Enable face culling.
7. Draw the geometry as lines using `glPolygonMode`.
8. Enable writes to the color buffer, disable face culling.
9. Set the stencil function to pass if the stencil value is 1.
10. Rendering a rectangle that fills the entire window (this will produce the silhouette image).
11. Draw the true edges of the geometry.
12. Enable writes to the depth buffer.

Since the algorithm uses an offset depth image it is susceptible to minor artifacts from the interaction of the lines and the depth image similar to those present when using `glPolygonOffset` for hidden line drawings.

7.5 Preventing Smooth Wide Line Overlap

When drawing a series of wide smoothed lines that overlap, such as an outline composed of a `GL_LINE_LOOP`, more than one fragment may be produced for a given pixel. Since smooth lines require enabling `GL_BLEND`, this may cause the pixel to appear brighter or darker than expected, as the fragments add more color to that pixel than in other locations.

An application may use a combination of the stencil test and alpha test to pass only the fragments that have the highest alpha, and therefore contribute the most color to a pixel. This technique uses repeated application of the alpha test to pass fragments with decreasing alpha, and uses the stencil test and buffer to mark where fragments previously passed. This has the effect of sorting fragments by alpha value.

```
glClear(GL_STENCIL_BUFFER_BIT);
glEnable(GL_STENCIL_TEST);
glEnable(GL_ALPHA_TEST);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_BLEND);
glStencilFunc(GL_NOTEQUAL, 1, 0xff);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
for(a = .98f; a >= 0.0f; a -= .02f) {
    glAlphaFunc(GL_GREATER, a);
    /* draw lines here */
}
```

Because this draws the line set repeatedly (50 times in this example), you should consider the alpha values likely to be used by your application and alter the loop appropriately.

For example, to improve performance by reducing the number of iterations, your application may favor higher alpha values by increasing the step size as the value in the loop decreases, or simply end the loop early.

On the other hand, if your application requires more accuracy, it is possible to iterate through every possible alpha value and pass only the fragments in each iteration that match each specific alpha value.

7.6 End Caps On Wide Lines

If wide lines form a loop, like a silhouette edge or the outline of a polygon, it may be necessary to fill regions where one line ends and another begins, to give the appearance of a rounded joint. Smoothed wide points may be applied at the ends of the line segments to form an end cap.

Use an algorithm like the one presented in Section 7.5 to avoid saturating pixels with the line and point color.

8 Blending and Compositing

OpenGL provides a rich set of blending operations which can be used to implement transparency, compositing, painting, and other effects. Rasterized fragments are linearly combined with pixels in the selected color buffers, clamped to 1.0 and then written to the color buffers. The `glBlendFunc` command selects the source and destination blend factors. The most frequently used factors are `GL_ZERO`, `GL_ONE`, `GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA`. OpenGL 1.1 specifies additive blending, but vendors have added extensions to allow other blending equations such as subtraction and reverse subtraction, and several of these extensions are standard commands in OpenGL 1.2, or are part of the "imaging subset" of OpenGL 1.2 (see Section 13.1.4).

Most OpenGL implementations use fixed point representations for color throughout the fragment processing path. The color component resolution is typically 5, 8, or 12 bits. Resolution problems usually show up when attempting to blend many images into the color buffer, for example, in some volume rendering techniques or multilayer composites. Some of these problems can be alleviated using the accumulation buffer instead, but the accumulation buffer does not provide the same flexibility for building up results and hardware accumulation buffer support is not as widely available as blending support.

OpenGL does not require that implementations support an alpha buffer ("destination alpha") for storing alpha values like the other color components. For many applications this is not a limitation, but there is a class of multipass operations where maintaining the current computed alpha value is necessary.

8.1 Compositing

The OpenGL blending operation does not directly implement the compositing operations described by Porter and Duff [79]. The difference is that in their compositing operations the colors are premultiplied by the alpha value and the resulting factors used to scale the colors are simplified after this scaling. It has been proposed that OpenGL be extended to include the ability to premultiply the source color values by alpha to better match the Porter and Duff operations. In the meantime, its certainly possible to achieve the same effect by computing the premultiplied values in the color buffer itself. For example, if there is an image in the color buffer, a new image can be generated which multiplies each color component by its alpha value and leaves the alpha value unchanged by performing a `glCopyPixels` operation with blending enabled and the blending function set to (`GL_SRC_ALPHA`,`GL_ZERO`). To ensure that the original alpha value is left intact, use the `glColorMask` command to disable updates to the alpha component during the copy operation.

8.2 Advanced Blending

OpenGL 1.1 only allows simple additive combinations of the source and destination color components during blending. Two ways in which the blending operations have been extended by vendors include the ability to blend with a constant color and the ability to use other blending equations. The blending color extension (`EXT_blend_color`) adds a constant RGBA color state variable which can be used as a blending factor in the blend equation. This capability can be very useful for implementing blends between two images without needing to specify the individual source and destination alpha components on a per pixel basis.

The blend equation extension (`EXT_blend_minmax`) provides the framework for specifying alternate blending equations. For example, in OpenGL 1.1, the accumulation buffer is the only mechanism which allows pixel values to be subtracted, but there is no easy method to include a per-pixel scaling factor such as alpha, so a subtractive blending equation has been implemented as an extension to 1.1 and is part of the imaging subset in OpenGL 1.2. Min and max functions are useful in image processing algorithms (e.g., for computing maximum intensity projections) and are also implemented as an extension to 1.1 and as part of the 1.2 imaging subset.

Operation	Action
GL_ACCUM	read from selected buffer, scale by value, then add into accumulation buffer
GL_LOAD	read from selected buffer, scale by value, then use image to replace contents of accumulation buffer
GL_RETURN	scale image by value, then copy into buffers selected for writing
GL_ADD	add value to R, G, B, and A components of every pixel in accumulation buffer
GL_MULT	clamp value to range -1 to 1, then scale R, G, B, and A components of every pixel in accumulation buffer.

Table 2: `glAccum` Operations

8.3 Painting

Two-dimensional painting applications can make interesting use of texturing and blending. An arbitrary image can be used as a paint brush, using blending to accumulate the contribution over time. The image source (paint brush) can be geometry or a pixel image. A texture mapped quad under an orthographic projection can be used in the same way as a pixel image and often more efficiently (when texture mapping is hardware accelerated).

An interesting way to implement the painting process is to precompute the effect of painting the entire image with the brush and then use blending to selectively expose the painted area as the brush passes over the area. This can be implemented efficiently with texturing by using the fully painted image as a texture map, blending the source image mapped on the brush with the current image stored in the color buffer. Use a geometric shape and translate the $< s, t >$ texture coordinates as the $< x, y >$ coordinates move across the image. The main advantage of this technique is that elaborate paint/brush combinations can be efficiently computed across the entire image all at once rather than performing localized computations in the area covered by the brush.

8.4 Blending with the Accumulation Buffer

The accumulation buffer is designed for combining multiple images. Instead of simply replacing pixel values with incoming pixel fragments, the fragments are scaled and then added to the existing pixel value. In order to maintain accuracy over many blending operations, the accumulation buffer has a higher number of bits per color component than a typical color buffer.

The accumulation buffer can be cleared like any other buffer. You can use `glClearAccum` to set the red, green, blue, and alpha components of its clear color. Clear the accumulation buffer by bitwise or'ing in the `GL_ACCUM_BUFFER_BIT` value to the parameter of the `glClear` command.

You can't render directly into the accumulation buffer. Instead you render into a selected color buffer, then use `glAccum` to accumulate that image into the accumulation buffer. The `glAccum` command reads from the currently selected read buffer. You can set the buffer you want it to read from using the `glReadBuffer` command.

The `glAccum` command takes two arguments, *op* and *value*. The possible settings for *op* are described in Table 2.

Since you must render to another buffer before accumulating, a typical approach to accumulating images is to render images to the back buffer some number of times, accumulating each image into the accumulation buffer. When the desired number of images have been accumulated, the contents of the accumulation buffer are copied into the back buffer, and the buffers are swapped. This way, only the final accumulated image is displayed.

Here is an example procedure for accumulating n images:

1. Call `glDrawBuffer(GL_BACK)` to render to the back buffer only.

2. Call `glReadBuffer(GL_BACK)` so that the accumulation buffer will read from the back buffer.

Note that the first two steps are only necessary if the application has changed the selected draw and read buffers. If the visual is double buffered, these settings are the default.

3. Clear the back buffer with `glClear`, then render the first image.
4. Call `glAccum(GL_LOAD, 1.f/n)`; this allows you to avoid a separate step to clear the accumulation buffer.
5. Alter the parameters of your image, and re-render it.
6. Call `glAccum(GL_ACCUM,1.f/n)` to add the second image into the first.
7. Repeat the previous two steps n - 2 more times...
8. Call `glAccum(GL_RETURN, 1.f)` to copy the completed image into the back buffer.

The accumulation buffer provides a way to take "multiple exposures" of a scene, while maintaining good color resolution. There are a number of image effects that can be implemented with the accumulation buffer to improve the realism of a rendered image [43, 70], including antialiasing, motion blur, soft shadows, and depth of field. To create these effects, render the image multiple times, making small, incremental changes to the scene position (or selected objects within the scene), and accumulate the results.

8.5 Blending Transitions

When generating real-time or interactive imagery, often the application may switch between different representations of an object. A different representation may be chosen which provides more detail or less detail, takes less time to render, or for a variety of other reasons. The two representations may not be similar enough to generate the same pixels on the screen, so the transition may generate an objectionable "pop" on the screen. The apparent discontinuity can be reduced by fading the old representation in and the new representation over a number of frames using blending. The new representation is rendered with `glBlendFunc(GL_SRC_ALPHA, GL_ONE)` and the old representation with `glBlendFunc(GL_ONE_MINUS_SRC_ALPHA, GL_ONE)`, varying alpha from 0 to 1 over a few frames.

8.6 The Stencil Buffer

The stencil buffer is like the depth and color buffers, except stencil pixels don't represent colors or depths, but have application-specific meanings. The stencil buffer isn't directly visible like the color buffer, but the bits in the stencil planes form an unsigned integer that affects and is updated by drawing commands, through the stencil function and the stencil operations. The stencil function controls whether a fragment is discarded or not by the stencil test, and the stencil operation determines how the stencil planes are updated as a result of that test [67].

Stencil buffer actions are part of OpenGL's fragment operations. Stencil testing occurs immediately after the alpha test, and immediately before the depth test. If `GL_STENCIL_TEST` is enabled, and stencil planes are available, the application can control what happens under three different scenarios:

1. The stencil test fails.
2. The stencil test passes, but the depth test fails.
3. Both the stencil and the depth test pass.

Comparison	Description of comparison test between reference and stencil value
`GL_NEVER`	always fails
`GL_ALWAYS`	always passes
`GL_LESS`	passes if reference value is less than stencil buffer
`GL_LEQUAL`	passes if reference value is less than or equal to stencil buffer
`GL_EQUAL`	passes if reference value is equal to stencil buffer
`GL_GEQUAL`	passes if reference value is greater than or equal to stencil buffer
`GL_GREATER`	passes if reference value is greater than stencil buffer
`GL_NOTEQUAL`	passes if reference value is not equal to stencil buffer

Table 3: Stencil Buffer Comparisons

Stencil Operation	Results of Operation on Stencil Values
`GL_KEEP`	stencil value unchanged
`GL_ZERO`	stencil value set to zero
`GL_REPLACE`	stencil value replaced by stencil reference value
`GL_INCR`	stencil value incremented
`GL_DECR`	stencil value decremented
`GL_INVERT`	stencil value bitwise inverted

Table 4: Stencil Buffer Operations

Whether a stencil operation for a given fragment passes or fails has nothing to do with the color or depth value of the fragment. The stencil operation is a comparison between the value in the stencil buffer for the fragment's destination pixel and the stencil reference value. A mask is bitwise AND-ed with the value in the stencil planes and with the reference value before the comparison is applied. The reference value, the comparison function, and the comparison mask are set by `glStencilFunc`. The comparison functions available are listed in Table 3.

Stencil function and *stencil test* are often used interchangeably in these notes, but the "stencil test" specifically means the application of the stencil function in conjunction with the stencil mask.

If the stencil test fails, the fragment is discarded (the color and depth values for that pixel remain unchanged) and the stencil operation associated with the stencil test failing is applied to that stencil value. If the stencil test passes, then the depth test is applied. If the depth test passes (or if depth testing is disabled or if the visual does not have a depth buffer), the fragment continues on through the pixel pipeline, and the stencil operation corresponding to both stencil and depth passing is applied to the stencil value for that pixel. If the depth test fails, the stencil operation set for stencil passing but depth failing is applied to the pixel's stencil value.

Thus, the stencil test controls which fragments continue towards the framebuffer, and the stencil operation controls how the stencil buffer is updated by the results of both the stencil test and the depth test.

The stencil operations available are described in Table 4. The `GL_INCR` and `GL_DECR` operations saturate so incrementing the maximum stencil value is still the maximum value and decrementing the value zero is still zero. Some implementations support the `EXT_stencil_wrap` extension which adds two new stencil operations, `GL_INCR_WRAP_EXT` and `GL_DECR_WRAP_EXT`. These operations "wrap" such that incrementing the maximum stencil value generates zero and decrementing zero generates the maximum value.

The `glStencilOp` call sets the stencil operations for all three stencil test results: stencil fail, stencil pass/depth buffer fail, and stencil pass/depth buffer pass.

Writes to the stencil buffer can be disabled and enabled per bit by `glStencilMask`. This allows an application to apply stencil tests without the results affecting the stencil values. Keep in mind, however, that the `GL_INCR` and `GL_DECR` operations operate on each stencil value as a whole, and may not operate as expected when the

OpenGL Implementation	Stencil Bits Supported
Most software implementations (Mesa, Microsoft OpenGL, SGI OpenGL for Windows)	8
3Dlabs Permedia II	1
SGI Indigo^2 Extreme	4
SGI Octane MXI	8
ATI Rage 128	8 (32-bit mode only)
NVIDIA RIVA TNT	8 (32-bit mode only)
SGI Onyx^2 InfiniteReality	1 or 8 (multisampled!)

Table 5: Stencil Bits Supported by Selected OpenGL Implementations

stencil mask is not all ones. Stencil writes can also be disabled by calling `glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP)`.

There are three other important ways of controlling and accessing the stencil buffer. Every stencil value in the buffer can be set to a desired value by calling `glClearStencil` and `glClear(GL_STENCIL_BUFFER_BIT)`. The contents of the stencil buffer can be read into system memory using `glReadPixels` with the format parameter set to `GL_STENCIL_INDEX`. The contents of the stencil buffer can also be set using `glDrawPixels`.

Different machines support different numbers of stencil bits per pixel. Table 5 lists the number of stencil bits supported by a few selected OpenGL implementations. Use `glGetIntegerv(GL_STENCIL_BITS, ...)` to see how many bits are available. If multiple stencil bits are available, `glStencilMask` and the mask argument to `glStencilFunc` can be used to divide up the stencil buffer into a number of different sections. This allows the application to store separate stencil values per pixel within the same stencil buffer.

The following sections describe how to use the stencil buffer in a number of useful multipass rendering techniques.

8.7 Compositing Images with Depth

Compositing separate images together is a useful technique for increasing the complexity of a scene [24]. Section 8.1 discusses algorithms for compositing two images together using alpha values to control how pixels are merged. One drawback of this method is that only simple visibility information can be expressed using mattes or masks. Using the stencil buffer, it is possible to merge images using the original depth information from the images. Both color and depth images can be independently saved to memory and later drawn to the screen using `glDrawPixels`. This is sufficient for 2D style composites, where objects are drawn on top of each other to create the final scene. To do true 3D compositing, it is necessary to use the color and depth values simultaneously, so that depth testing can be used to determine which surfaces are obscured by others.

The stencil buffer can be used for true 3D compositing in a two pass operation. The color buffer is disabled for writing, the stencil buffer is cleared, and the saved depth values are copied into the framebuffer. Depth testing is enabled, insuring that only depth values that are closer to the original can update the depth buffer. `glStencilOp` is called to set a stencil buffer bit if the depth test passes.

The stencil buffer now contains a mask of pixels that were closer to the view than the pixels of the original image. The stencil function is changed to accomplish this masking operation, the color buffer is enabled for writing, and the color values of the saved image are drawn to the frame buffer.

This technique works because the fragment operations, in particular the depth test and the stencil test, are part of both the geometry and imaging pipelines in OpenGL. Here is the technique in more detail. It assumes that both the depth and color values of an image have been saved to system memory, and are to be composited using depth testing to an image in the framebuffer:

1. Clear the stencil buffer using `glClear`, or'ing in `GL_STENCIL_BUFFER_BIT`.

2. Disable the color buffer for writing with `glColorMask`.
3. Set stencil values to 1 when the depth test passes by calling `glStencilFunc(GL_ALWAYS, 1, 1)`, and `glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE)`.
4. Ensure depth testing is set; `glEnable(GL_DEPTH_TEST)`, `glDepthFunc(GL_LESS)`.
5. Draw the depth values to the framebuffer with `glDrawPixels`, using `GL_DEPTH_COMPONENT` for the format argument.
6. Set the stencil buffer to test for stencil values of 1 with `glStencilFunc(GL_EQUAL, 1, 1)` and `glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP)`.
7. Disable the depth testing with `glDisable(GL_DEPTH_TEST)`.
8. Draw the color values to the framebuffer with `glDrawPixels`, using `GL_RGBA` as the format argument.

At this point, both the depth and color values will have been merged, using the depth test to control which pixels from the saved image would update the framebuffer. Compositing can still be problematic when merging images with coplanar polygons.

This process can be repeated to merge multiple images. The depth values of the saved image can be manipulated by changing the values of `GL_DEPTH_SCALE` and `GL_DEPTH_BIAS` with `glPixelTransfer`. This technique could allow you to squeeze the incoming image into a limited range of depth values within the scene.

9 Antialiasing

Aliasing refers to the jagged edges and other rendering artifacts commonly associated with computer-generated drawings. It is caused by the presence of higher frequency renderings than can be represented by the pixel samples. Lines are much more susceptible to aliasing problems because every pixel drawn is part of an edge while most pixels of polygon models are in the middle where there are no high frequences. More detailed explanations of why this is so are available in [68], [69], [59], and [20].

9.1 Line and Point Antialiasing

Line and point antialiasing should be considered separately from polygon antialiasing since the techniques are usually quite different. Mathematically, a line is infinitely thin. Attempting to compute the percentage of a pixel covered by an infinitely thin object would be impossible, so generally one of the following two methods is used:

1. The line is modeled as a long, thin, single-pixel-wide quadrilateral. The percentage of pixel coverage is computed for each pixel touching the line and this coverage percentage is used as the alpha value for blending.
2. The line is modeled as an infinitely thin transparent glowing object. This method treats a line as if drawn on a vector stroke display where the display draws a line by deflecting the electron beam as opposed to a raster display that moves the beam in horizontal scans and varies the beam intensity. This approach requires the implementation to compute the effective shape of an electron beam as it moves across the CRT phosphors.

To antialias points or lines in OpenGL, you need to enable antialiasing by calling `glEnable` and passing in `GL_POINT_SMOOTH` or `GL_LINE_SMOOTH`, as appropriate. You can also provide a quality hint by calling `glHint`. The hint parameter can be `GL_FASTEST` to indicate that the most efficient option should be chosen, `GL_NICEST` to indicate the highest quality option should be chosen, or `GL_DONT_CARE` to indicate no preference.

When antialiasing is enabled, OpenGL computes an alpha value representing either the fraction of each pixel that is covered by the line or point or the beam intensity for the pixel as a function of the distance of the pixel center from the line center. The setting of the `GL_LINE_SMOOTH` and the `GL_POINT_SMOOTH` hints determine how accurate the calculation is when rendering lines and points, respectively. When the hint is set to `GL_NICEST`, a larger filter function may be applied causing more fragments to be generated and rendering to slow down.

No matter which line antialiasing method is used in your particular version of OpenGL, you can approximate either by choosing the right blend equation. The important point to remember is that antialiased lines and points are a form of *transparent primitive*. This requires blending to be enabled so that the incoming pixel fragment will be combined with the value already in the framebuffer, depending on the alpha value.

The best approximation of a one-pixel-wide quadrilateral is achieved by setting the blending factors to `GL_SRC_ALPHA` (source) and `GL_ONE_MINUS_SRC_ALPHA` (destination). To best approximate the lines of a stroke display, use `GL_ONE` for the destination factor. Note that this second blend equation only works well on a black background and does not produce good results when drawn over bright objects.

As with all transparent primitives, antialiased lines and points should not be drawn until all opaque objects have been drawn first. Depth buffer testing should remain enabled, but depth buffer updating should be disabled using `glDepthMask(GL_FALSE)`. Antialiased lines drawn with full depth buffering enabled produce incorrect line crossings and can result in significantly worse rendering artifacts than with antialiasing disabled when a lot of lines are drawn close together.

If the destination blend mode is set to `GL_ONE_MINUS_SRC_ALPHA` there may be visible order dependent rendering artifacts if the antialiased primitives are not drawn in back to front order. There are no such order dependent problems with a setting of `GL_ONE`, however. It is best to pick the method that best suits your particular application.

Incorrect monitor gamma settings are much more likely to become apparent with antialiased lines than shaded polygons. Broadcast television uses a gamma value of 2.22. The gamma value needed to correct most color CRTs is usually between 2.0 and 2.6. Some workstation manufacturers use values as low as 1.6 to enhance the perceived contrast of rendered images even though it produces a definite intensity nonlinearity in displayed images. Signs of insufficient gamma are "roping" of lines and moire patterns where many lines come together. Too much gamma produces a "washed out" appearance.

Antialiasing in color index mode is trickier because you have to load the color map correctly to get primitive edges to blend with the background color. When antialiasing is enabled, the last four bits of the color index indicate the coverage value. Thus, you need to load sixteen contiguous colormap locations with a color ramp ranging from the background color to the object's color. This technique only works well when drawing wireframe images, where the lines and points typically need to be blended with a constant background. If the lines and/or points need to be blended with background polygons or images, RGBA rendering should be used.

9.2 Polygon Antialiasing

Antialiasing the edges of filled polygons is similar to antialiasing points and lines. However, antialiasing polygons in color index mode isn't practical since object intersections are more prevalent and you really need to use OpenGL blending to get decent results.

To enable polygon antialiasing call `glEnable` with `GL_POLYGON_SMOOTH`. This causes pixels on the edges of the polygon to be assigned fractional alpha values based on their coverage. Also, if you want, you can supply a value for `GL_POLYGON_SMOOTH_HINT`.

In order to get the polygons blended correctly when they overlap, you need to sort the polygons in front to back order in eye space. *This method does not work without sorting and requires an alpha buffer.* Before rendering, disable depth testing, enable blending and set the blending factors to `GL_SRC_ALPHA_SATURATE` (source) and `GL_ONE` (destination). The final color will be the sum of the destination color and the scaled source color; the scale factor is the smaller of either the incoming source alpha value or one minus the destination alpha value. This means that for a pixel with a large alpha value, successive incoming pixels have little effect on the final color because one minus the destination alpha is almost zero.

Since the accumulated coverage is stored in the color buffer, destination alpha is required for this algorithm to work. Thus you must request a visual or pixel format with destination alpha. OpenGL does not require implementations to support a destination alpha buffer so visual selection may fail.

9.3 Multisampling

Multisampling is an antialiasing method that provides high quality results. It is available as an OpenGL extension from at least one vendor. In this technique additional subpixel storage is maintained as part of the color, depth and stencil buffers. Instead of using alpha for coverage, coverage masks are computed to help maintain sub-pixel coverage information for all pixels. Current implementations support four, eight, and sixteen samples per pixel. The method allows for full scene antialiasing at a modest performance penalty but a more substantial storage penalty (since sub-pixel samples of color, depth, and stencil need to be maintained for every pixel). This technique does not entirely replace the methods described above, but is complementary. Antialiased lines and points using alpha coverage can be mixed with multisampling as well as the accumulation buffer antialiasing method.

9.4 Antialiasing With Textures

You can also antialias points and lines using the filtering provided by texturing using alpha textures. Simply create an image of a circle where the alpha values are one in the center and go to zero as you move from the center out to an edge. The alpha texel values would then be used to blend the point or rectangle fragments with the pixel values already in the framebuffer.

For example, to draw antialiased points, create a texture image containing a filled circle with a smooth (antialiased) boundary. Then draw a textured polygon at the point making sure that the center of the texture is aligned with the point's coordinates and using the texture environment `GL_MODULATE`. This method has the advantage that any point shape may be accommodated simply by varying the texture image.

A similar technique can be used to draw antialiased line segments of any width. The texture image is a filtered circle as described above. Instead of a line segment, a texture mapped rectangle, whose width is the desired line width, is drawn centered on and aligned with the line segment. If line segments with round ends are desired, these can be added by drawing an additional textured rectangle on each end of the line segment.

9.5 Antialiasing with Accumulation Buffer

Accumulation buffers can be used to antialias a scene without having to depth sort the primitives before rendering. A supersampling technique is used where the entire scene is offset by small, subpixel amounts in screen space, and accumulated. The jittering can be accomplished by modifying the transforms used to represent the scene.

One straightforward jittering method is to modify the projection matrix, adding small translations in x and y. Care must be taken to compute the translations so that they shift the scene the appropriate amount in window coordinate space. Fortunately, computing these offsets is straightforward. To compute a jitter offset in terms of pixels, divide the jitter amount by the dimension of the object coordinate scene, then multiply by the appropriate viewport dimension. The example code fragment below shows how to calculate a jitter value for an orthographic projection; the results are applied to a translate call to modify the modelview matrix:

```
void ortho_jitter(GLfloat xoff, GLfloat yoff)
{
    GLint viewport[4];
    GLfloat ortho[16];
    GLfloat scalex, scaley;

    glGetIntegerv(GL_VIEWPORT, viewport);
    /* this assumes that only a glOrtho() call has been
    applied to the projection matrix */
    glGetFloatv(GL_PROJECTION_MATRIX, ortho);

    scalex = (2.f/ortho[0])/viewport[2];
    scaley = (2.f/ortho[5])/viewport[3];
    glTranslatef(xoff * scalex, yoff * scaley, 0.f);
}
```

If the projection matrix wasn't created by calling `glOrtho` or `gluOrtho2D`, then you will need to use the viewing volume extents (right, left, top, bottom) to compute `scalex` and `scaley` as follows:

```
    GLfloat right, left, top, bottom;

    scalex = ((right-left)/viewport[2];
    scaley = ((top-bottom)/viewport[3];
```

The code is very similar for jittering a perspective projection. In this example, we jitter the frustum itself:

```
void frustum_jitter(GLdouble left, GLdouble right,
                    GLdouble bottom, GLdouble top,
                    GLdouble near, GLdouble far,
                    GLdouble xoff, GLdouble yoff)
{
```

Count	Values
2	{0.25, 0.75}, {0.75, 0.25}
3	{0.5033922635, 0.8317967229}, {0.7806016275, 0.2504380877}, {0.2261828938, 0.4131553612}
4	{0.375, 0.25}, {0.125, 0.75}, {0.875, 0.25}, {0.625, 0.75}
5	{0.5, 0.5}, {0.3, 0.1}, {0.7, 0.9}, {0.9, 0.3}, {0.1, 0.7}
6	{0.4646464646, 0.4646464646}, {0.1313131313, 0.7979797979}, {0.5353535353, 0.8686868686}, {0.8686868686, 0.5353535353}, {0.7979797979, 0.1313131313}, {0.2020202020, 0.2020202020}
8	{0.5625, 0.4375}, {0.0625, 0.9375}, {0.3125, 0.6875}, {0.6875, 0.8125}, {0.8125, 0.1875}, {0.9375, 0.5625}, {0.4375, 0.0625}, {0.1875, 0.3125}
9	{0.5, 0.5}, {0.1666666666, 0.9444444444}, {0.5, 0.1666666666}, {0.5, 0.8333333333}, {0.1666666666, 0.2777777777}, {0.8333333333, 0.3888888888}, {0.1666666666, 0.6111111111}, {0.8333333333, 0.7222222222}, {0.8333333333, 0.0555555555}
12	{0.4166666666, 0.625}, {0.9166666666, 0.875}, {0.25, 0.375}, {0.4166666666, 0.125}, {0.75, 0.125}, {0.0833333333, 0.125}, {0.75, 0.625}, {0.25, 0.875}, {0.5833333333, 0.375}, {0.9166666666, 0.375}, {0.0833333333, 0.625}, {0.583333333, 0.875}
16	{0.375, 0.4375}, {0.625, 0.0625}, {0.875, 0.1875}, {0.125, 0.0625}, {0.375, 0.6875}, {0.875, 0.4375}, {0.625, 0.5625}, {0.375, 0.9375}, {0.625, 0.3125}, {0.125, 0.5625}, {0.125, 0.8125}, {0.375, 0.1875}, {0.875, 0.9375}, {0.875, 0.6875}, {0.125, 0.3125}, {0.625, 0.8125}

Table 6: Sample Jittering Values

```
    GLfloat scalex, scaley;
    GLint viewport[4];

    glGetIntegerv(GL_VIEWPORT, viewport);
    scalex = (right - left)/viewport[2];
    scaley = (top - bottom)/viewport[3];

    glFrustum(left - xoff * scalex,
              right - xoff * scalex,
              top - yoff * scaley,
              bottom - yoff * scaley,
              near, far);
}
```

The jittering values you choose should fall in an irregular pattern. In other words, it is undesirable to have the sample points line up in *any* direction. This reduces aliasing artifacts by making them "noisy". Selected subpixel jitter values, organized by the number of samples needed, are taken from the *OpenGL Programming Guide*, and are shown in Table 6. (Note that some of these patterns are a little more regular horizontally and vertically than is optimal.)

Using the accumulation buffer, you can easily trade off quality and speed. For higher quality images, simply increase the number of scenes that are accumulated. Although it is simple to antialias the scene using the accumulation buffer, it is much more computationally intensive and probably slower than the polygon antialiasing method described above.

10 Lighting Techniques

This section discusses varies ways of improving and refining the lighting of your scenes using OpenGL.

10.1 Phong Shading

10.1.1 Phong Highlights with Texture

One of the problems with the OpenGL lighting model is that specular radiance is computed before textures are applied in the normal pipeline sequence. To achieve more realistic looking results, specular highlights should be computed and added to image after the texture has been applied. This can be accomplished by breaking the shading process into two passes. In the first pass diffuse radiance is computed for each surface and then modulated by the texture colors to be applied to the surface and the result written to the color buffer. In the second pass the specular highlight is computed for each polygon and added to the image in the framebuffer using a blending function which sums 100% of the source fragment and 100% of the destination pixels. For this particular example we will use an infinite light and a local viewer. The steps to produce the image are as follows:

1. Define the material with appropriate diffuse and ambient reflectance and zero for the specular reflectance coefficients.
2. Define and enable lights.
3. Define and enable texture to be combined with diffuse lighting.
4. Define modulate texture environment.
5. Draw lit, textured object into the color buffer.
6. Define new material with appropriate specular and shininess and zero for diffuse and ambient reflectance.
7. Disable texturing, enable blending, set the blend function to `GL_ONE`, `GL_ONE`.
8. Draw the specular-lit, non-textured geometry.
9. Disable blending.

10.1.2 Improved Highlight Shape

This implements the basic algorithm, but the Gouraud shaded specular highlight still leaves something to be desired. We can improve on the specular highlight by using environment mapping to generate a higher quality highlight. We generate a sphere map consisting only of a Phong highlight [78] and then use the `GL_SPHERE_MAP` texture coordinate generation mode to generate texture coordinates which index this map. For each polygon in the object, the reflection vector is computed at each vertex. Since the coordinates of the vector are interpolated across the polygon and used to lookup the highlight, a much more accurate sampling of the highlight is achieved compared to interpolation of the highlight value itself. The sphere map image for the texture map of the highlight can be computed by rendering a highly tessellated sphere lit with only a specular highlight using the regular OpenGL pipeline. Since the direction of the light relative to the view direction is effectively encoded in the texture map, the texture map needs to be recomputed whenever the light or viewer position is changed. Sphere mapping assumes that the view direction is constant (infinite viewer) and the environment (light) direction is infinitely far away, so the highlight does not need to be changed when the object moves.

The nine step method outlined above needs minor modifications to incorporate the new lighting method:

6. Disable lighting.

7. Load the sphere map texture, enable the sphere map texgen function.
8. Enable blending, set the blend function to `GL_ONE`, `GL_ONE`.
9. Draw the unlit, textured geometry with vertex colors set to 1.0.
10. Disable texgen, disable blending.

With a little work the technique can be extended to handle multiple light sources. OpenGL 1.2 includes new functionality which enables the per-vertex lighting computation to compute a specular contribution separate from the ambient, diffuse, and emissive contributions and adds this specular contribution in after the application of the texture environment. Since this contribution is calculated per-vertex and interpolated it solves the specular-after-texture problem, but it does provide any additional improvement in the shape or quality of the highlight, so the above technique remains useful for improving the highlight quality.

10.1.3 Spotlight Effects using Projective Textures

The projective texture technique described earlier can be used to generate a number of interesting illumination effects. One of the possible effects is spotlight illumination. The OpenGL lighting model already includes a spotlight illumination model, providing control over the cutoff angle (spread of the cone), the exponent (concentration across the cone), direction of the spotlight, and attenuation as a function of distance. The OpenGL model typically suffers from undersampling of the light. Since the lighting model is only evaluated at the vertices and the results are linearly interpolated, if the geometry being illuminated is not sufficiently tessellated incorrect illumination contributions are computed. This typically manifests itself by a dull appearance across the illuminated area or irregular or poorly defined edges at the perimeter of the illuminated area. Since the projective method samples the illumination at each pixel the undersampling problem is eliminated.

Similar to the Phong highlight method, a suitable texture map must be generated. The texture is an intensity map of a cross-section of the spotlight's beam. The same type of exponent parameter used in the OpenGL model can be incorporated or a different model entirely can be used. If 3D textures are available the attenuation due to distance can be approximated using a 3D texture in which the intensity of the cross-section is attenuated along the r-dimension. When geometry is rendered with the spotlight projection, the r coordinate of the fragment is proportional to the distance from the light source.

In order to determine the transformation needed for the texture coordinates, it is easiest to think about the case of the eye and the light source being at the same point. In this instance the texture coordinates should correspond to the eye coordinates of the geometry being drawn. The simplest method to compute the coordinates (other than explicitly computing them and sending them to the pipeline from the application) is to use an `GL_EYE_LINEAR` texture generation function with an `GL_EYE_PLANE` equation. The planes simply correspond to the vertex coordinate planes (e.g., the s coordinate is the distance of the vertex coordinate from the y-z plane, etc.). Since eye coordinates are in the range [-1.0, 1.0] and the texture coordinates need to be in the range [0.0, 1.0], a scale and translate of 0.5 is applied to s and t using the texture matrix. A perspective spotlight projection transformation can be computed using `gluPerspective` and combined into the texture transformation matrix. The transformation for the general case when the eye and light source are not in the same position can be computed by incorporating into the texture matrix the inverse of the transformations used to move the light source away from the eye position.

With the texture map available, the method for rendering the scene with the spotlight illumination is as follows:

1. Initialize the depth buffer.
2. Clear the color buffer to a constant value which represents the scene ambient illumination.
3. Draw the scene with depth buffering enabled and color buffer writes disabled.
4. Load and enable the spotlight texture, set the texture environment to `GL_MODULATE`.

5. Enable the texgen functions, load the texture matrix.
6. Enable blending and set the blend function to `GL_ONE`, `GL_ONE`.
7. Disable depth buffer updates and set the depth function to `GL_EQUAL`.
8. Draw the scene with the vertex colors set to 1.0.
9. Disable the spotlight texture, texgen and texture transformation.
10. Set the blend function to `GL_DST_COLOR`.
11. Draw the scene with normal illumination.

There are three passes in the algorithm. At the end of the first pass the ambient illumination has been established in the color buffer and the depth buffer contains the resolved depth values for the scene. In the second pass the illumination from the spotlight is accumulated in the color buffer. By using the `GL_EQUAL` depth function, only visible surfaces contribute to the accumulated illumination. In the final pass the scene is drawn with the colors modulated by the illumination accumulated in the first two passes to arrive at the final illumination values.

The algorithm does not restrict the use of texture on objects, since the spotlight texture is only used in the second pass and only the scene geometry is needed in this pass. The second pass can be repeated multiple times with different spotlight textures and projections to accumulate the contributions of multiple light sources.

There are a couple of considerations that also should be mentioned. Texture projection along the negative line-of-sight of the texture (back projection) can contribute undesired illumination. This can be eliminated by positioning a clip plane at the near plane of the line-of-site. Also, OpenGL encourages but does not guarantee pixel exactness when various modes are enabled or disabled. This can manifest itself in undesirable ways during multipass algorithms. For example, enabling texture coordinate generation may cause fragments with different depth values to be generated compared to the case when texture coordinate generation is not enabled. This problem can be overcome by re-establishing the depth buffer values between the second and third pass. This is done by redrawing the scene with color buffer updates disabled and the depth buffering configured the same as for the first pass. Also, use a texture wrap mode of `GL_CLAMP` to keep the spotlight pattern from repeating. When using a linear texture filter, use a black texel border to avoid clamping artifacts or, if available, use the `GL_CLAMP_TO_EDGE` wrap mode.

It is also possible to render the entire scene in a single pass. If none of the objects in the scene are textured, the complete image could be rendered once, if the ambient illumination can be summed with spotlight illumination while the objects are rendered. Some vendors have added an additive texture environment function as an extension which makes this operation feasible. A cruder method that works in OpenGL 1.1 involves illuminating the scene using normal OpenGL lighting, using the spotlight texture to modulate the scene brightness.

10.1.4 Phong Shading by Adaptive Tessellation

Phong highlights can also be approached with a modeling technique. The surface can be adaptively tessellated until the difference between $(\vec{H} \cdot \vec{N})^n$ terms on triangle vertices drops below a predetermined value. The advantage of this technique is that it can be done as a separate pre-processing step. The disadvantage is that it increases the complexity of the modeled object. This can be costly if:

- The model will have to be clipped by a large number of user-defined clipping planes.
- The model will have tiled textures applied to it.
- The performance of the application/system is already triangle limited.

10.2 Light Maps

A light map is a texture map applied to a material to simulate the effect of a local light source. Like specular highlights, it can be used to improve the appearance of local light sources without resorting to excessive tessellation of the objects in the scene. A excellent example of an application using lightmaps is the interactive PC game Quake^{TM}. This game uses light maps to simulate the effects of local light sources, both stationary and moving, to great effect.

Using lightmaps usually requires a multipass algorithm, unless the objects being mapped are untextured. A texture simulating the light's effect on the object is created, then applied to one or more objects in the scene. Appropriate texture coordinates are generated, and texture transformations can be used to position the light, and create moving or changing light effects. Multiple light sources can be generated with a combination of more complex texture maps and/or more passes to the algorithm.

Light maps are often luminance textures, which are applied to the object using `GL_MODULATE` as the value for `GL_TEXTURE_ENV_MODE`. Colored lights can also be simulated by using an RGB texture.

Light maps can often produce satisfactory lighting effects at lower resolutions than normal textures. It is often not necessary to produce mipmaps; choosing `GL_LINEAR` for the minification and magnification filters is sufficient. Of course, the minimum quality of the lighting effect is a function of the intended application.

10.2.1 2D Texture Light Maps

A 2D light map is a texture map applied to the surfaces of a scene, modulating the intensity of the surfaces to simulate the effects of a local light. If the surface is already textured, then applying the light map becomes a multipass operation, modulating the intensity of a surface detail texture.

A 2D light map can be generated analytically, creating a bright spot in luminance or color values that drops off appropriately with increasing distance from the light center. As with other lighting equations, a quadratic drop off, modified with linear and constant terms can be used to simulate a variety of lights, depending on the area of the emitting source.

Since generating new textures takes time and consumes valuable texture memory, it is a good strategy to create a few canonical light maps, based on intensity drop-off characteristics and color, then use them for a number of different lights by transforming the texture coordinates. If the light source is isotropic, then simple translations and scales can be used to position the light appropriately on the surface, while scales can be used to adjust the size of the lighting effect, simulating different sizes of lights and distance from the lighted surface.

In order to apply a light map to a surface properly, the position of the light in the scene must be projected onto each surface of interest. This position shows where the bright spot will be. The perpendicular distance of the light from the surface can be used to adjust the bright spot size and brightness. One approach is to generate texture coordinates, orienting the generating planes with each surface of interest, then translating and scaling the texture matrix to position the light on the surface. This process is repeated for every surface affected by the light.

In order to repeat this process for multiple lights (without resorting to a multilight lightmap) or to light textured surfaces, the lighting must be done as a series of passes. This can be done two ways. The more straightforward way is to blend the entire scene. The other way is to blend together the surface texture and light maps to create a texture for each surface. This texture will represent the contributions of the surface texture and all lightmaps affecting its surface. The merged texture is then applied to the surface. Although more involved, the second method produces a higher quality result.

For each surface:

1. Transform the surface so that it is perpendicular to the direction of view (maximize its visible surface). Scale the image so that its area in pixels matches the desired size of the final texture.

2. Render the transformed surface into the frame buffer (this can be done in the back buffer). If it is textured, render it with the surface texture.
3. Re-render the surface, using the appropriate light map. Adjust the `GL_EYE_PLANE` equations and the texture transform to position the light correctly on the surface. Use the appropriate blend function.
4. Repeat the previous step with each light visible to the surface.
5. Copy the image into a texture using `glCopyTexImage2D`.
6. When you've created textures for all lit surfaces, render the scene using the new textures.

Since switching between textures must be done quickly, and lightmap textures tend to be small, use texture objects to switch between different light maps and surface textures to improve performance.

With either approach, the blending is a modulation of the colors of the existing texture. This can be done by rendering with the blend function (`GL_ZERO`, `GL_SRC_COLOR`). If the light map is composed of luminance values than the individual destination color components will be scaled equally, if the light map represents a colored light, then the color components of the destination will be scaled by the red, green, and blue components of the light map texel values.

Note that each modulation pass attenuates the surface color. The results will become increasingly dim. If surfaces require a large number of lights, the dynamic range of light maps can be compressed to avoid excessive darkening. Instead of ranging from 1.0 (full light) to 0.0 (no light), They can range from 1.0 (full light) to 0.5 or 0.75 (no light). The no light value can be adjusted as a function of the number of lights in the scene.

Here are the steps for using 2D Light Maps:

1. Create the 2D light data. "Canonical lights" can be defined at the center of the texture, with the intensity dropping off in a realistic fashion towards the edges. In order to avoid artifacts, make sure the intensity of the light field is the same at all the edges of the texture volume.
2. Define a 2D texture, using `GL_REPEAT` for the wrap values in s, t, and r. Minification and magnification should be `GL_LINEAR` to make the changes in intensity smoother. For performance reasons, make this texture a texture object.
3. Render the scene without the lightmap, using surface textures as appropriate.
4. For each light in the scene:
   (a) For each surface in the scene:
      i. Cull surfaces that cannot "see" the current light.
      ii. Find the plane of the surface.
      iii. Align the `GL_EYE_PLANE` for `GL_s` and `GL_t` with the surface plane.
      iv. Scale and translate the texture coordinates to position and size the light on the surface.
      v. Render the surface using the appropriate blend function and lightmap texture.

An alternative to simple light maps is to use projective textures to draw light sources. This is a good approach when doing spotlight effects. It's not as useful for isotropic light sources, since you'll have to tile your projections to make the light shine in all directions. See the projective texture description in Section 10.1.2 and in Section 6.16 for more details.

10.2.2 3D Texture Light Maps

3D Textures can also be used as light maps. One or more light sources are represented in 3D data, then the 3D texture is applied to the entire scene. The main advantage of using 3D textures for light maps is that it's easy to calculate the proper texture coordinates. The textured light source can be positioned globally with the appropriate texture transformations then the scene is rendered, using `glTexGen` to generate the proper s, t, and r coordinates.

The light source can be moved by changing the texture matrix. The resolution of the light field is dependent on the texture resolution.

A useful approach is to define a canonical light field in 3D texture data, then use it to represent multiple lights at different positions and sizes by applying texture translations and scales to shift and resize the light. Multiple lights can be simulated by accumulating the results of each light source on the scene.

To ensure that the light source can be shifted easily, set `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T`, and `GL_TEXTURE_WRAP_R_EXT` to `GL_REPEAT`. Then the light can be shifted to any location in the scene. Be sure that the texel values in the light map are the same at all boundaries of the texture; otherwise you'll be able to see the edges of the texture as vertical and horizontal "shadows" in the scene.

Although it is uncommon, some types of light fields would be very hard to do without 3D textures. A complex light source, whose brightness and range varies as a function of distance from the light source could be best done with a 3D texture. An example might be a "disco ball" effect where a light source has beams emanating out from the center, with some beams shining farther than others. A complex light source could be made more impressive by combining light maps with volume visualization techniques. For example the light beams could be made visible in fog.

The light source itself can be a simple piece of geometry textured with the rest of the scene. Since it is at the source of the textured light, it will be textured brightly.

For better realism, good lighting effects should be combined with the shadowing techniques described in Section 11.4.

Procedure:

1. Create the 3D light data. A "canonical light" can be defined at the center of the texture volume, with the intensity dropping off in a realistic fashion towards the edges. In order to avoid artifacts, make sure the intensity of the light field is the same at all the edges of the texture volume.
2. Define a 3D texture, using `GL_REPEAT` for the wrap values in S, t, and R. Minification and magnification should be `GL_LINEAR` to make the changes in intensity smoother.
3. Render the scene without the lightmap, using surface textures as appropriate.
4. Define planes in eye space so that `glTexGen` will cause the texture to span the visible scene.
5. If you have textured surfaces, adding a lightmap becomes a multipass technique. Use the appropriate blending function to modulate the surface color.
6. Render the image with the light map, and texgen enabled. Use the appropriate texture transform to position and scale the light source correctly.
7. Repeat steps 1-2 and 4-6 for each light source.

There are disadvantages to using 3D light maps:

- 3D textures are not widely supported yet, so your application will not be as portable.
- 3D textures use a lot of texture memory. 2D textures are more efficient for light maps.

10.3 Gloss Maps

Surfaces whose shininess varies, like wet marble or wet paper or fabrics that are smoothed only in places, can be modeled with *gloss maps*. A gloss map is a texture which modulates the contribution from specular lighting, so that some points on the surface reflect less specular light than others.

This technique can be implemented using a multipass process in which the diffuse, ambient, and emissive lighting components are drawn and then the specular lighting component is added using blending.

First, draw the surface normally with specular lighting disabled. This can be performed by setting the specular material value to zero, for example, in OpenGL, by `glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specularColorArray)`, and then drawing the surface normally.

Second, draw the surface with the specular color restored and the diffuse, ambient, and emissive colors set to zero, and with a texture encoding the gloss map. The texture can be a one component alpha texture or a two or four component texture with the maximum value set for the luminance or color components.

Typically the gloss map value is stored directly in the alpha component, with zero indicating no contribution from the specular component and one indicating the full specular reflection is to be added. Draw the second pass with blending enabled, adding the new pixel color modulated by the alpha value from the texture. (In OpenGL this behavior is configured with `glBlendFunc(GL_SRC_ALPHA, GL_ONE)`). The meaning of the alpha component may also be reversed if the applications takes care to reverse the frame buffer blending operation as well.

The second pass can be drawn either with the depth test set to pass when the framebuffer depth equals the depth of the surface, or the initial pass can set the stencil buffer where the depth test passed and the second pass drawn where the stencil buffer was set.

10.4 Other Lighting Models

Up to this point we have largely discussed the Phong lighting model. The diffuse and specular terms for a single light are given by the following equation:

$$d_m d_l \max(\vec{N} \cdot \vec{L}, 0) + s_m s_l \max(\vec{H} \cdot \vec{N}, 0)^n$$

Section 10.1.1 discusses the use of sphere mapping to replace the OpenGL per-vertex specular illumination computation with one performed at each pixel. The specular contribution in the texture map is computed using the Phong formulation above. However, the Phong model can be substituted with other bi-directional reflectance functions to achieve other lighting effects. Since the texture coordinates are computed with a sphere mapping function, the resulting texture mapping operation accurately approximates view-dependent specular reflectance distributions.

One improvement that can be made is to add a *Fresnel* reflection term, F_λ,[45] to the specular equation:

$$d_m d_l \max(\vec{N} \cdot \vec{L}, 0) + F_\lambda s_m s_l \max(\vec{H} \cdot \vec{N}, 0)^n$$

The Fresnel term specifies the ratio the amount of reflected light to the amount of transmitted (refracted) light. It is a function of the angle of incidence, θ_i, the angle of refraction θ_t and the material properties of the object (dielectric, metal, etc. as described in Section 10.8). The effect of the Fresnel term is to attenuate light as a function of its incident and reflected directions as well as its wavelength. Light is hardly reflected from dielectrics such as glass at normal incidence, for example, while being almost totally reflected at glancing angles. This attenuation is independent of wavelength. The absorption of metals, on the other hand, can be a function of the wavelength in, for instance, copper and gold. At glancing angles, the light color is unaltered in reflection, but at normal incidence the light is modulated by the color of the metal.

Since the sphere map serves as a table which is indexed by the the reflection vector, the Fresnel effects can be included in the environment map by simply computing the specular equation with the Fresnel term to modulate and shift the color. This can be performed as a post-processing step on an existing environment map by computing the Fresnel reflection coefficient at each angle of incidence and modulating the sphere map. Reflection, refraction and sphere mapping are discussed in more detail in Section 11.1. Other bi-directional reflectance functions can be encoded in a sphere map in a similar fashion.

10.5 Global Illumination

The lighting models described thus far have been relatively simple. The subtleties of real lighting are often captured using a global illumination model. Global illumination models using radiosity or ray tracing are generally too computationally complex to perform in real-time. However, if the objects and light sources comprising the environment are static it is possible to perform the global illumination calculations as a preprocessing step and then display the results interactively. Such an approach is both practical and useful for applications such as architectural walkthroughs. The technique is typically employed for diffuse illumination solutions since view-independent (ideal) diffuse illumination can be represented as a single value (color) at each object vertex.

In [99] Walter, et. al. describe a method for rendering global illumination solutions which contain view-independent directionally variant lighting effects using the specular term in the OpenGL lighting model to approximate the directionally varying lighting information and the emissive term to approximate the directionally invariant illumination (i.e., diffuse illumination). In this method, a set of OpenGL lights are treated as a set of basis functions which are summed together while the object is rendered to yield a more general directional distribution. The OpenGL light parameters such as position or intensity coefficients have no relationship to the light sources in the original model, but instead serve as a compact representation for the directional illumination of an object. Each rendered object has its own set of lights which are called *virtual lights*.

The method works on a global illumination solution which stores a number of samples of the directionally varying illumination at each object vertex. The parameters for the virtual lights of a particular object are determined using a fitting procedure consisting of a number of heuristics. The main idea is to produce a set of solutions for a number of specular exponent values and then choose the exponent value which minimizes the mean-squared error using a least squares method. A solution at a given exponent value is determined as follows:

1. Choose a specular exponent value.
2. Find the vertex on the object with the largest directional radiance.
3. Choose a light direction to align the specular lobe with this brightest direction.
4. Choose an intensity coefficient to match the radiance at the point on the object.
5. Compute the specular contribution at other points on the object and subtract from the radiance.
6. Repeat steps 2-5 using updated object radiance until all lights have been used.
7. At each vertex compute the specular and emission coefficients using a least squares fit.

Once the lighting parameters have been determined the model is rendered using the `glLight` and `glMaterial` commands to set the directional light parameters and specular exponent for each object and the `glMaterial` command to set the specular reflectance and and emitted intensity at each vertex. The rendering speed for the model is limited by the geometric complexity of the model and the ability of the OpenGL implementation to deal with multiple light sources and material changes at each vertex. Rendering performance may be improved by rendering in multiple passes to limit the number of active lights or the number of material parameter changes in each pass. For example, using `glColorMaterial` and `glColor` to change only the emitted intensity or specular reflectance in each pass and framebuffer blending to sum the results together.

10.6 Bump Mapping with Textures

Bump Mapping Bump mapping [9], like texture mapping, is a technique to add more realism to synthetic images without adding a lot of geometry. Texture mapping adds realism by attaching images to geometric surfaces. Bump mapping adds per-pixel surface relief shading, increasing the apparent complexity of the surface by perturbing the surface normal. Surfaces that have a patterned roughness are good candidates for bump mapping. Examples include oranges, strawberries, stucco, wood, etc.

An intuitive representation of surface bumpiness is formed by a 2D height field array, or *bump map*. This bump map is defined by the scalar difference $F(u,v)$ between the flat surface $P(u,v)$ and the desired bumpy surface $P'(u,v)$ along the normal N at each point u, v. Typically the function P is modeled separately as polygons or parametric patches and F is modeled as a 2D image using a painting program or other image processing tool.

Rather than subdivide the surface $P'(u,v)$ into regions that are locally flat, observe that the shading perturbations on such a surface depend more on perturbations in the surface normal than on the position of the surface itself. A technique perturbing only the surface normal at shading time achieves similar results without the processing burden of subdividing geometry. (Note that this technique does not perturb shadows from other surfaces falling on the bumps or shadows from bumps on the same surface, so such shadows will retain their flat appearance.)

The normal vector $\vec{N'}$ at u, v can be calculated by the cross product of the partial derivatives of P' in u and v. (The notational simplification P'_u is used here to mean the partial derivative of P' with respect to u, sometimes written $\frac{\partial P'}{\partial u}$.) The chain rule can be applied to the partial derivatives to yield the following expression of P'_u P'_v in terms of P, F, and derivatives of F:

$$P'_u = P_u + F_u(\frac{N}{|N|}) + F\frac{\partial \frac{N}{|N|}}{\partial u}$$

$$P'_v = P_v + F_v(\frac{N}{|N|}) + F\frac{\partial \frac{N}{|N|}}{\partial v}$$

If F is assumed to be sufficiently small, the final terms of each of the previous expressions can be approximated by zero:

$$P'_u = \ P_u + F_u(\frac{N}{|N|})$$

$$P'_v = \ P_v + F_v(\frac{N}{|N|})$$

Expanding the cross product $P'_v \times P'_u$ gives the following expression for N':

$$N' = (P_u + F_u(\frac{N}{|N|})) \times (P_v + F_v(\frac{N}{|N|}))$$

Which evaluates to:

$$N' = P_u \times P_v + \frac{F_u(N \times P_v)}{|N|} + \frac{F_v(N \times P_u)}{|N|} + \frac{F_u F_v (N \times N)}{|N|^2}$$

Because $P_u \times P_v$ yields the normal N and $N \times N$ yields 0, we can further simplify the expression for N' as follows:

$$N' = N + \frac{F_u(N \times P_v)}{|N|} + \frac{F_v(N \times P_u)}{|N|}$$

The values F_u and F_v are easily computed through forward differencing from the 2D bump map, and P_u and P_v can be computed either directly from the surface definition or from forward differencing applied to the surface parameterization.

Approximating Bump Mapping Using Texture Typically, bump mapping is implemented in custom rendering software on the host, as few hardware systems implement bump map evaluation directly. It may be possible to directly bump map an object covering a small number of pixels in software and composite the image into the non-bump-mapped scene using techniques discussed elsewhere in these notes, and retain interactive frame rates. However, the prohibitive cost of texture access and vector normalization make hardware assistance attractive.

The following sections present a technique for using texture maps to approximate bump mapping without requiring a custom renderer [1] [77]. This multipass algorithm is an extension and refinement of texture embossing [86].

Tangent Space Recall that the bump map normal $\vec{N'}$ is formed by $\vec{Pu} \times \vec{Pv}$. Assume that the surface P is coincident with the XY plane and that changes in u and v correspond to changes in X and Y, respectively. Then F can be substituted for P', resulting in the following expression for the vector N':

$$N' = \begin{pmatrix} -\frac{\partial F}{\partial u} \\ -\frac{\partial F}{\partial v} \\ 1 \end{pmatrix}$$

In order to evaluate the lighting equation, N' must be normalized. If the displacements in the bump map are restricted to small values, however, the length of N' will be so close to one as to be approximated by one. Then N' itself can be substituted for $\vec{N}$ without normalization. If the diffuse intensity component $\vec{N} \cdot \vec{L}$ of the lighting equation is evaluated with the value presented above for N', the result is the following:

$$N' \cdot L = -\frac{\partial F}{\partial u} L_x - \frac{\partial F}{\partial v} L_y + L_z \tag{1}$$

This expression requires the surface to lie in the XY plane and that the u and v parameters change in X and Y, respectively. Most surfaces, however, will have arbitrary locations and orientations in space. In order to use this simplification to perform bump mapping, the view direction $\vec{V}$, and light source direction $\vec{L}$ are transformed into *tangent space*.

Tangent space has 3 axes, $\vec{T}$, $\vec{B}$ and $\vec{N}$. The tangent vector, $\vec{T}$, is parallel to the direction of increasing s on the surface. The normal vector, $\vec{N}$, is perpendicular to the surface. The binormal, $\vec{B}$, is perpendicular to both $\vec{N}$ and $\vec{T}$, and like $\vec{T}$, lies in the plane tangent to the surface. These vectors form a coordinate system that is attached to and varies over the surface.

The light source is transformed into tangent space at each vertex of the polygon. To find the tangent space vectors at a vertex, use the vertex normal for $\vec{N}$ and find the tangent axis $\vec{T}$ by finding the vector direction of increasing s in the object's coordinate system. The direction of increasing t may also be used. Find $\vec{B}$ by computing the cross product of $\vec{N}$ and $\vec{T}$. These unit vectors form the transformation shown below:

$$\begin{pmatrix} x' \\ y' \\ z` \\ w` \end{pmatrix} = \begin{pmatrix} T_x & T_y & T_z & -V_x \\ B_x & B_y & B_z & -V_y \\ N_x & N_y & N_z & -V_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{2}$$

This transformation brings coordinates into tangent space, where the plane tangent to the surface lies in the $X-Y$ plane, and the normal to the surface coincides with the Z axis. Note that the tangent space transformation varies for vertices representing a curved surface, and so this technique makes the approximation that curved surfaces are flat and the tangent space transformation is interpolated from vertex to vertex.

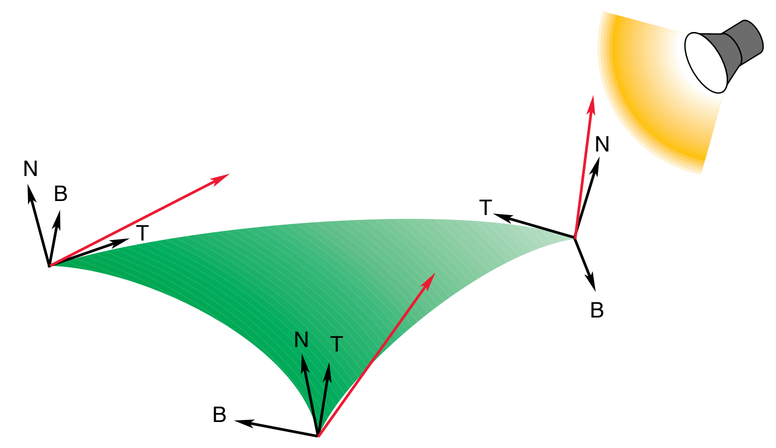


Figure 47. Tangent Space Defined at Polygon Vertices

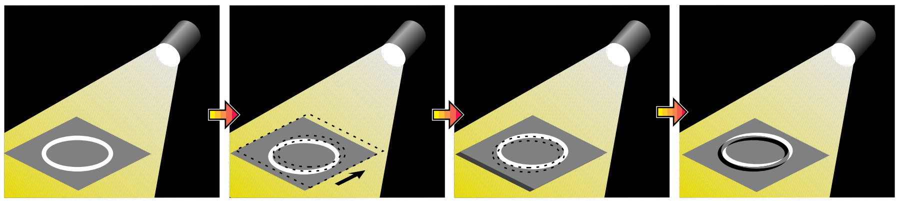


Figure 48. Bump Mapping: Shift and Subtract Image

Forward Differencing The first derivative of the height values of the bump map in a given direction s', t' can be approximated by the following process:

1. Render the bump map texture.
2. Shift the texture coordinates at the vertices by s', t'.
3. Re-render the bump map texture, subtracting from the first image.

Consider a one dimensional bump map for simplicity. The map only varies as a function of s. Assuming that the height values of the bump map can be represented as a height function $F(s)$, then the three step process above would be the following: $F(s) - F(s + \Delta s)$. If the delta is one texel in s, then the resulting texture coordinate is $F(s) - F(s + \frac{1}{w})$, where w is the width of the texture in texels. This operation implements a forward difference of F, which would approximate the first derivative of F if F was continuous.

In the two dimensional case, the height function is $F(s, t)$, and performing the forward difference in the direction of s', t' evaluates the derivative of $F(s, t)$ in the direction s', t'. This technique is also used to create embossed images.

This operation provides the values used for the first two addends shown in Equation 1. In order to provide the third addend of the dot product, the process needs to compute and add the transformed z component of the light

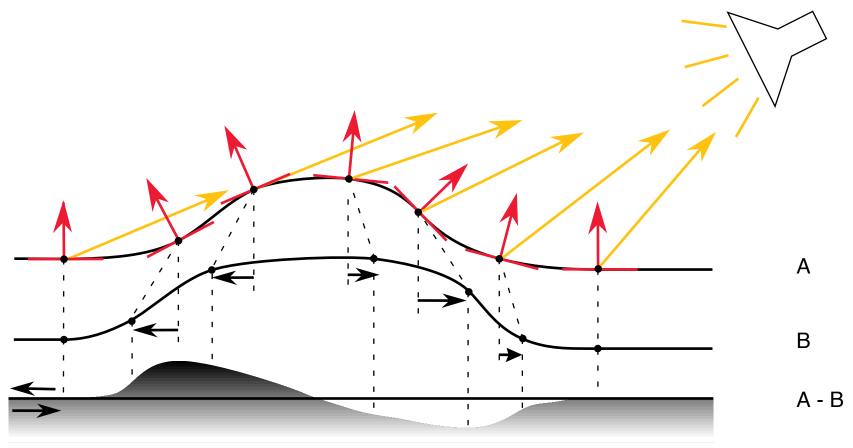


Figure 49. Shifting a Bump Map to Perform Forward Differencing

vector. The tangent space transform in Equation 2 implies that the transformed z component of L is simply the inner product of the vertex normal and the light vector, $N \cdot L$. Therefore, the z component can be computed using OpenGL to evaluate the diffuse lighting term at each vertex. This computation is performed as a second pass, adding to the previous results.

The steps for diffuse bump mapping are the following:

1. Render the polygon with the bump map textured on it. Since the bump map modifies the polygon color, you can get the diffuse color you want by coloring the polygon with k_d. Lighting is disabled.
2. Find $\vec{N}$, $\vec{T}$ and $\vec{B}$ at each vertex.
3. Use the vectors to create a transformation.
4. Use the matrix to rotate the light vector $\vec{L}$ into tangent space.
5. Use the rotated x and y components of $\vec{L}$ to shift the s and t texture coordinates at each polygon vertex.
6. Re-render the bump map textured polygon using the shifted texture coordinates.
7. Subtract the second image from the first.
8. Render the polygon smooth shaded with lighting enabled and texturing disabled.
9. Add this image to result.

Using the accumulation buffer can improve the accuracy of this technique. The bump mapped objects in the scene are rendered with the bump map, re-rendered with the shifted bump map and accumulated with a negative weight, then re-rendered again using Gouraud shading and no bump map texture, accumulated normally.

The process can also be extended to find bump mapped specular highlights. The process is repeated using the halfway vector ($\vec{H}$) instead of the light vector. The halfway vector is computed by averaging the light and viewer vectors $\frac{\vec{L}+\vec{V}}{2}$. The combination of the forward difference of the bump map in the direction of the tangent space $\vec{H}$ and the z component of $\vec{H}$ approximate $N \cdot H$. Here are the steps for computing $(\vec{N} \cdot \vec{H})$:

1. Render the polygon with the bump map textured on it.
2. Find $\vec{N}$, $\vec{T}$ and $\vec{B}$ at each vertex.
3. Use the vectors to create a rotation matrix.
4. Use the matrix to rotate the halfway vector $\vec{H}$ into tangent space.
5. Use the rotated x and y components of $\vec{H}$ to shift the s and t texture coordinates at each polygon vertex.
6. Re-render the bump map textured polygon using the shifted texture coordinates.
7. Subtract the second image from the first.
8. Render the polygon Gouraud shaded with no bump map texture, this time use $\vec{H}$ instead of $\vec{L}$. Use a polygon whose color is equal to the specular color you want, k_s.

The result must be raised to the shininess exponent before blending.

One technique for performing this exponential is to use the texture color table extension to perform a table lookup on the results of $(\vec{H} \cdot \vec{N})$. Invoke `glColorTableSGI` with `GL_TEXTURE_COLOR_TABLE_SGI` as its target, then enable `GL_TEXTURE_COLOR_TABLE_SGI`. Copy the image calculated above from framebuffer memory into texture, using `glCopyTexImage2D`. Finally, apply a projective texture transformation or calculate texture coordinates so that the specular component is mapped onto the object with the same screen coordinates in which it was drawn. Pixels copied from other objects drawn will not fall on the bump mapped object and will be discarded. It is a good idea to copy only the rectangular screen region that bounds the bump-mapped object if the application supports this. If the texture color table extension is not available, it may be possible to use `glReadPixels`, `glTexSubImage2D`, apply the exponent on the host, and still retain interactivity for small regions.

Another possibility is to use `glCopyPixels` with a color table configured with `glPixelMap`. The bump mapped object can be applied to the stencil buffer to create a mask so that only the pixels covered by the bump mapped object are applied to the framebuffer.

Finally, use blending during the application of the specular component to add to the diffuse component and complete the lighting equation.

Improving Quality The previous technique renders the entire scene multiple times. If very high quality is important, the texture itself can be processed separately, then applied to the scene as a final step. The previous technique yields lower quality results where the texture is less perpendicular to the line of sight in the image, due to the object geometry. If the texture is processed before being applied to the image, we avoid this problem.

To process the texture separately, the vertices of the object must be mapped to a square grid. The rest of the steps are the same, because the relationship between light source and the vertex normals hasn't changed. When the new texture map has been created, copy it back into texture memory, and use it to render the object.

Blending If you choose not to use the accumulation buffer, acceptable results can be obtained by blending. Because of the subtraction step, you'll have to remap the color values to avoid negative results. Since the image values range from 0 to 1, the range of values after subtraction can be -1 (0 - 1) to 1 (1 - 0).

Scale and bias the bump map values to remap the results to the 0 to 1 range. Once you've made all three passes, it is safe to remap the values back to their original 0 to 1 range. This scaling and biasing, combined with fewer bits of color precision, make this method inferior to using the accumulation buffer.

Bumps on Surfaces Facing Away From the Light Because this algorithm doesn't take surfaces facing away from lights into account, the forward differencing calculation will produce "lights" on the surface even when no light is falling on the surface. Use the result of $\vec{L} \cdot \vec{N}$ to scale the shift so that the bump effect tapers off slowly as the surface becomes more oblique to the light direction. Empirically, adding a small bias (.3 in the authors' experiments) to the dot product (and clamping the result) is more visibly pleasing because the bumps appear to taper off *after* the surface has started facing away from the light, as would actually happen for a displaced surface.

Limitations Although this technique does closely approximate bump mapping, there are limitations that impact its accuracy.

Bump Map Sampling The bump map height function is not continuous, but is sampled into the texture. The resolution of the texture affects how faithfully the bump map is represented. Increasing the size of the bump map texture can improve the sampling of the high frequency height components.

Texture Resolution The shifting and subtraction steps produce the directional derivative. Since this is a forward differencing technique, the highest frequency component of the bump map increases as the shift is made smaller. As the shift is made smaller, more demands are made of the texture coordinate precision. The shift can become smaller than the texture filtering implementation can handle, leading to noise and aliases effects. A good starting point is to size the shift components so their vector magnitude is a single texel.

Surface Curvature The tangent coordinate axes are different at each point on a curved surface. This technique approximates this by finding the tangent space transforms at each vertex. Texture mapping interpolates the different shift values from each vertex across the polygon. For polygons with very different vertex normals, this approximation can break down. A solution would be to subdivide the polygons until their vertex normals are parallel to within some error limit.

Maximum Bump Map Slope The bump map normals used in this technique are good approximations if the bump map slope is small. If there are steep tangents in the bump map, the assumption that the perturbed normal is length one becomes inaccurate, and the highlights appear too bright. This can be corrected by creating a fourth pass, using a modulating texture derived from the original bump map. Each value of the texel is one over the length of the perturbed normal: $1/\sqrt{\frac{\partial f}{\partial u}^2 + \frac{\partial f}{\partial v}^2 + 1}$

10.7 Bump Mapped Reflections

To the authors' knowledge, at least one hardware platform available at the time of writing supports a further embellishment to bump mapping: bump mapped reflections.

If the bump map is stored as displacements to the normal ($\frac{\partial F}{\partial u}$ and $\frac{\partial F}{\partial u}$), rather than a height field, the displacements can be used as offsets added to the texture coordinates used in a second texture. This second texture represents the lighting environment, and can be the environment mapped approximation to phong lighting discussed in Section 10.1.1, or an environment map approximating reflections from the surface as discussed in Section 11.2.1.

10.8 Choosing Material Properties

OpenGL provides a full lighting model to help produce realistic objects. The library provides no guidance, however, on finding the proper lighting material parameters to simulate specific materials. This section categorizes common materials, provides some guidance for choosing representative material properties, and provides a table of material properties for common materials.

10.8.1 Modeling Material Type

Material properties are modeled with the following OpenGL parameters:

`GL_AMBIENT` How ambient light reflects from the material surface. This is an RGBA color vector. The magnitude of each component indicates how much the light of that component is being reflected.

`GL_DIFFUSE` How diffuse reflection from light sources reflect from the material surface. This is an RGBA color vector. The magnitude of each component indicates how much the light of that component is being reflected.

`GL_SPECULAR` How specular reflection from a light source reflects from the material. This is an RGBA color vector. The magnitude of each component indicates how much the light of that component is being reflected.

`GL_EMISSION` How much of what color is being emitted from this object. This is an RGBA color vector. The magnitude of each component indicates how much light of that component is glowing from the material. Since this parameter is only useful for glowing objects, we'll ignore it in this section.

`GL_SHININESS` How mirror-like the specular reflection is from this material. This is a single integer. The larger the number, the more rapidly the specular reflection drops off as the viewing angle diverges from the reflection vector.

For lighting purposes, materials can be described by the type of material, and the smoothness of its surface. Material type is simulated by the relationship between color components of the `GL_AMBIENT`, `GL_DIFFUSE` and `GL_SPECULAR` parameters. Surface smoothness is simulated by the overall magnitude of the `GL_AMBIENT`, `GL_DIFFUSE` and `GL_SPECULAR` parameters, and the value of `GL_SHININESS`. As the magnitude of these components get closer to one, and the `GL_SHININESS` value increases, the material appears to have a smoother surface.

For lighting purposes, material type can be divided into four categories: dielectrics, metals, composites, and other materials.

Dielectrics These are the most common category. These are non-conductive materials, such as plastic or wood, which don't have free electrons. The result is that dielectrics have relatively low reflectivity, and have a reflectivity that is independent of light color. Because they don't interact with the light much, many dielectrics are transparent. The ambient, diffuse and specular colors tend to be the same.

Powdered dielectrics tend to look white because of the high surface area between the dielectric and the surrounding air. Because of this high surface area, they also tend to reflect diffusely.

Metals Metals are conductive and have free electrons. As a result, metals are opaque and tend to be very reflective, and their ambient, diffuse, and specular colors tend to be the same. How the free electrons are excited by light at different wavelengths determines the color of the metal. Materials like steel and nickel have nearly the same response over all visible wavelengths, resulting in a grayish reflection. Copper and gold, on the other hand, reflect long wavelengths more strongly than short ones, giving them their reddish and yellowish colors.

The color of light reflected from metals is also a function of incident and exiting light directions. This can't be modeled accurately with the OpenGL lighting model, compromising the metallic look of objects. However, a modified form of environment mapping (such as the OpenGL sphere mapping) can be used to approximate the proper visual effect.

Composite Materials Common composites, like plastic and paint, are composed of a dielectric binder with metal pigments suspended in them. As a result, they combine the reflective properties of metals and dielectrics. Their specular reflection is dielectric, their diffuse reflection is like metal.

Other Materials Other materials that don't fit into the above categories are materials such as thin films, and other exotics.

10.8.2 Modeling Material Smoothness

As mentioned before, the apparent smoothness of a material is a function of how strongly it reflects and the size of the specular highlight. This is affected by the overall magnitude of the `GL_AMBIENT`, `GL_DIFFUSE` and `GL_SPECULAR` parameters, and the value of `GL_SHININESS`. Here are some heuristics that describe useful relationships between the magnitudes of these parameters:

1. The spectral color of the `GL_AMBIENT` and `GL_DIFFUSE` parameters should be the same.
2. The magnitudes of `GL_DIFFUSE` and `GL_SPECULAR` should sum to a value close to one. This helps prevent color value overflow.
3. The value of `GL_SHININESS` should increase as the magnitude of `GL_SPECULAR` approaches one.

No promise is made that these relationships, or the values in Table 7 will provide a perfect imitation of a given material. The empirical model used by OpenGL emphasizes performance, not physical exactness.

For an excellent description of material properties, see [45].

10.9 Anisotropic Lighting

The per-vertex lighting model used in OpenGL assumes that the surface has microscopic facets that are uniformly distributed in any direction on the surface. That is to say, they assume isotropic lighting behavior.

Some surfaces have a directional grain, made from facets that are formed with a directional bias, like the grooves formed by sanding or machining. These surfaces demonstrate *anisotropic* lighting, which depends on the rotation of the surface around the normal to the surface. At normal distances, the viewer does not see the facets or grooves, but rather sees the overall lighting effect from such a surface. Some everyday surfaces that have anistropic lighting behavior are hair, satin Christmas tree ornaments, brushed alloy wheels, CDs, cymbals in a drum kit, and vinyl records.

Heidrich and Seidel present a technique in [50] for rendering surfaces with anisotropic lighting, based on the scientific visualization work of Zöckler et al [92]. The technique uses 2D texturing to provide a lighting solution based on a "most significant" normal to a surface at a point.

The algorithm uses a surface model with infinitely thin scratches or threads that run across the surface. The tangent vector $\vec{T}$ defined per-vertex can be thought of as the direction of brush strokes, grooves, or threads. An infinitely thin thread can be considered to have an infinite number of surface normals distributed in the plane perpendicular to $\vec{T}$, as shown in Figure 50. In order to fully model the light reflected from these normals, the lighting equation would need to be integrated over the normal plane.

Rather than integrate the lighting equation, the technique makes the assumption that the most significant light reflection is from the surface normal $\vec{N}$ with the maximum dot product with the light vector $\vec{L}$ as seen in Figure 51.

The diffuse and specular lighting factors for a point based on the view vector $\vec{V}$, normal $\vec{N}$, light reflection vector $\vec{R}$, light direction $\vec{L}$, and shininess exponent s are shown below:

$$I_{diffuse} = L \cdot N$$

Material	GL_AMBIENT	GL_DIFFUSE	GL_SPECULAR	GL_SHININESS
Brass	0.329412	0.780392	0.992157	27.8974
	0.223529	0.568627	0.941176	
	0.027451	0.113725	0.807843	
	1.0	1.0	1.0	
Bronze	0.2125	0.714	0.393548	25.6
	0.1275	0.4284	0.271906	
	0.054	0.18144	0.166721	
	1.0	1.0	1.0	
Polished	0.25	0.4	0.774597	76.8
Bronze	0.148	0.2368	0.458561	
	0.06475	0.1036	0.200621	
	1.0	1.0	1.0	
Chrome	0.25	0.4	0.774597	76.8
	0.25	0.4	0.774597	
	0.25	0.4	0.774597	
	1.0	1.0	1.0	
Copper	0.19125	0.7038	0.256777	12.8
	0.0735	0.27048	0.137622	
	0.0225	0.0828	0.086014	
	1.0	1.0	1.0	
Polished	0.2295	0.5508	0.580594	51.2
Copper	0.08825	0.2118	0.223257	
	0.0275	0.066	0.0695701	
	1.0	1.0	1.0	
Gold	0.24725	0.75164	0.628281	51.2
	0.1995	0.60648	0.555802	
	0.0745	0.22648	0.366065	
	1.0	1.0	1.0	
Polished	0.24725	0.34615	0.797357	83.2
Gold	0.2245	0.3143	0.723991	
	0.0645	0.0903	0.208006	
	1.0	1.0	1.0	
Pewter	0.105882	0.427451	0.333333	9.84615
	0.058824	0.470588	0.333333	
	0.113725	0.541176	0.521569	
	1.0	1.0	1.0	

Table 7: Parameters for Common Materials

Material	GL_AMBIENT	GL_DIFFUSE	GL_SPECULAR	GL_SHININESS
Silver	0.19225 0.19225 0.19225 1.0	0.50754 0.50754 0.50754 1.0	0.508273 0.508273 0.508273 1.0	51.2
Polished Silver	0.23125 0.23125 0.23125 1.0	0.2775 0.2775 0.2775 1.0	0.773911 0.773911 0.773911 1.0	89.6
Emerald	0.0215 0.1745 0.0215 0.55	0.07568 0.61424 0.07568 0.55	0.633 0.727811 0.633 0.55	76.8
Jade	0.135 0.2225 0.1575 0.95	0.54 0.89 0.63 0.95	0.316228 0.316228 0.316228 0.95	12.8
Obsidian	0.05375 0.05 0.06625 0.82	0.18275 0.17 0.22525 0.82	0.332741 0.328634 0.346435 0.82	38.4
Pearl	0.25 0.20725 0.20725 0.922	1.0 0.829 0.829 0.922	0.296648 0.296648 0.296648 0.922	11.264
Ruby	0.1745 0.01175 0.01175 0.55	0.61424 0.04136 0.04136 0.55	0.727811 0.626959 0.626959 0.55	76.8
Turquoise	0.1 0.18725 0.1745 0.8	0.396 0.74151 0.69102 0.8	0.297254 0.30829 0.306678 0.8	12.8
Black Plastic	0.0 0.0 0.0 1.0	0.01 0.01 0.01 1.0	0.50 0.50 0.50 1.0	32
Black Rubber	0.02 0.02 0.02 1.0	0.01 0.01 0.01 1.0	0.4 0.4 0.4 1.0	10

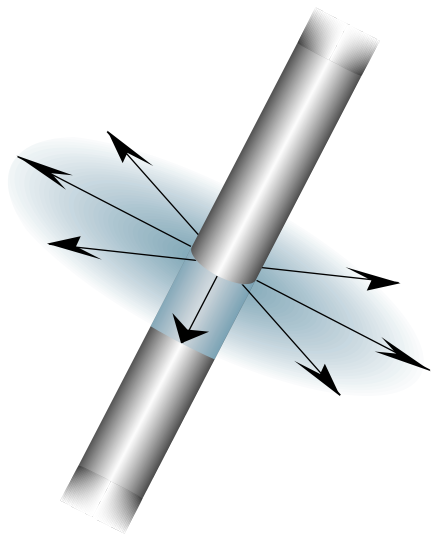


Figure 50. Normals to a Fiber

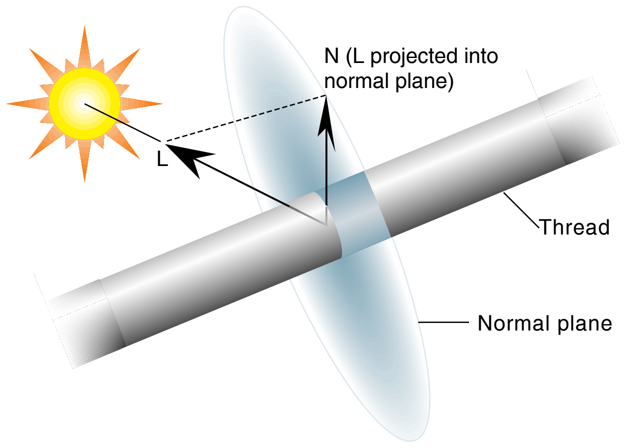


Figure 51. Projecting Light Vector to Maximize Lighting Contribution

$$I_{specular} = (V \cdot R)^s$$

In order to avoid calculating $\vec{N}$ and $\vec{R}$, the following substitutions allow the lighting calculation at a point on a fiber to be evaluated with only $\vec{L}$, $\vec{V}$, and the fiber tangent $\vec{T}$ (anisotropic bias):

$$N \cdot L = \sqrt{1 - (L \cdot T)^2}$$
$$V \cdot R = \sqrt{1 - (L \cdot T)^2} * \sqrt{1 - (V \cdot T)^2} - (L \cdot T)(V \cdot T)$$

If $\vec{V}$ and $\vec{L}$ are stored in the first two rows of a transformation matrix, and $\vec{T}$ is transformed by this matrix, the result is a vector containing $\vec{L} \cdot \vec{T}$ and $\vec{V} \cdot \vec{T}$. After applying this transformation, $L \cdot T$ is computed as s and $V \cdot T$ is computed as t, as shown in Equation 3 A scale and bias must also be included in the matrix in order to bring the dot product range $[-1, 1]$ into the range $[0, 1)$. The resulting texture coordinates can be used to index a texture storing the precomputed lighting equation.

$$\frac{1}{2}\begin{pmatrix} L_x & L_y & L_z & 1 \\ V_x & V_y & V_z & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}\begin{pmatrix} T_x \\ T_y \\ T_z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{2}(L \cdot T) + 1 \\ \frac{1}{2}(V \cdot T) + 1 \\ 0 \\ 1 \end{pmatrix} \tag{3}$$

If the further simplifications are made that the viewing vector is constant (infinitely far away) and that the light direction is also constant, then the results of this transformation can be used to index a 2D texture to evaluate the lighting equation based solely on providing $\vec{T}$ at each vertex.

The application will need to fill the texture with the results of the lighting equation (shown in Equation 17 in Appendix C.2) with the s and t coordinates scaled and biased back to the range $[-1, 1]$ and evaluated in the equations above to compute $N \cdot L$ and $V \cdot R$.

A transformation pipeline typically transforms surface normals into eye space by premultiplying by the inverse transpose of the viewing matrix. If the anisotropic bias $\vec{T}$ is defined in model space, it is necessary to query or precompute the current modeling transformation and concatenate the inverse transpose of that transformation with the transformation matrix computed above.

Because result of this lookup is not the complete anisotropic computation but rather the "most significant" component of it, it may be necessary to raise the diffuse and specular lighting factors used in the lighting computation to a large fractional power. (Since those factors will be less than one, a fractional power will increase the factors.) This may result in a more visually acceptable image.

OpenGL's texture matrix (`glMatrixMode(GL_TEXTURE)`) and vertex texture coordinate (`glTexCoord`) can be used to perform the texture coordinate computation directly. The transformation is stored in the texture matrix and $\vec{T}$ is transmitted using `glTexCoord` directly.

Keep in mind, however, that there is no normalization step in the OpenGL texture coordinate generation system. If the modeling matrix is concatenated as mentioned previously, the coordinates must be transformed and normalized before transmission.

Because the anisotropic lighting approximation given does not take self-shadowing into account, the texture color will also need to be modulated with a saturated directional light. This will clamp the lighting contributions to zero on parts of the surface facing away from the light.

This technique uses per-vertex texture coordinates to encode the anisotropic direction, so it also suffers from the same per-vertex lighting artifacts in the isotropic lighting model. In addition, if a local lighting or viewing model is desired, the application must calculate L or V, compute the entire anisotropic lighting contribution, and apply it as a vertex color, which frees the texture stage for another use.

Because a single texture provides all the lighting components up front, changing any of the colors used in the lighting model requires recalculating the texture. If two textures are used, either in a system with multiple texture units or with multipass, the diffuse and specular components may be separated and stored in two textures, and either texture may be modulated by the material or vertex color in order to alter the diffuse or specular base color separately without altering the maps. This can be used, for example, in a database containing a precomputed radiosity solution stored in the per-vertex color. In this way, the diffuse color can still depend on the orientation of the viewpoint relative to the anisotropic bias but only changes within the maximum color calculated by the radiosity solution.

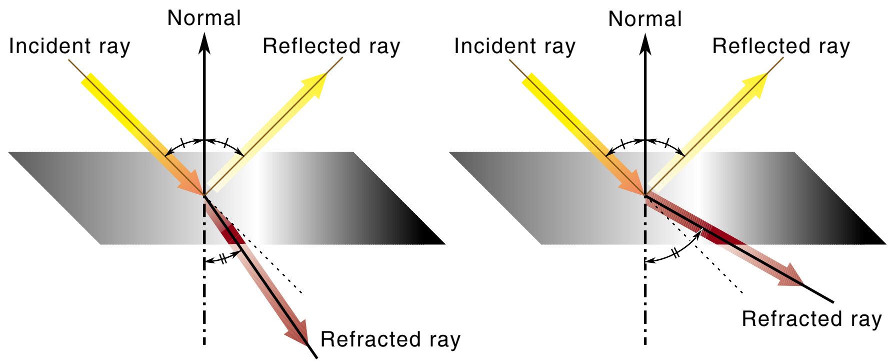


Figure 52. Reflection and Refraction: Medium Below has Higher Index of Refraction

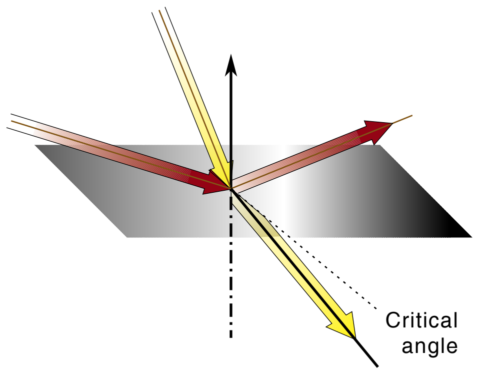


Figure 53. Total Internal Reflection

11 Scene Realism

11.1 Reflections and Refractions

In both rendering and interactive computer graphics, substantial effort has been devoted to the modeling of reflected and refracted light. This is not surprising – almost all the light perceived in the world is reflected. This section describes several ways to create the effects of reflection and refraction using OpenGL beginning with a very brief review of the relevant physics. Pointers to more detailed descriptions are provided.

From elementary physics, the angle of reflection of a ray is equal to the angle of incidence of the ray (Figure 52). This property is known as the *Law of Reflection* [21]. The reflected ray lies in the plane defined by the incident ray and the surface normal.

Refraction is defined as the “change in the direction of travel as light passes from one medium to another” [21]. This change in direction is caused by the difference in the speed of light traveling through the two media. The refractivity of a material is characterized by the *index of refraction* of the material, or the ratio of the speed of light in the material to the speed of light in a vacuum [21].

The direction of a light ray after it passes from one medium to another is computed from the direction of the incident ray, the normal of the surface at the intersection of the incident ray, and the indices of refraction of the two materials. The behavior is shown in Figure 52. The first medium through which the ray passes has an index of refraction n_1 and the second has an index of refraction n_2. The angle of incidence, Θ_1, is the angle between the incident ray and the surface normal. The refracted ray forms the angle Θ_2 with the normal. The incident and refracted rays are coplanar. The relationship between the angle of incidence and the angle of refraction is stated as *Snell's Law*[21]:

$$n_1 \cos \Theta_1 = n_2 \cos \Theta_2 \tag{4}$$

If $n_1 > n_2$ (light is passing from a more refractive material to a less refractive material), past some critical angle the incident ray will be bent so far that it will not cross the boundary. This phenomenon is known as *total internal reflection* and is illustrated in Figure 53 [21].

When a ray hits a surface, some light is reflected off the surface and some is transmitted. The weighting of the transmitted and reflected light is determined by the *Fresnel equations*.

More details about reflection and refraction can be gleaned from most college physics books. For more details on the reflection and transmission of light from a computer graphics perspective, consult one of several general computer graphics books or books on radiosity or ray tracing [17], [32], [45].

Accelerated Reflection and Refraction Directly calculating surface physics effects like reflection and refraction using an algorithm such as ray tracing can be expensive, and rendering almost immediately stops being interactive as scene complexity increases. It is useful to define techniques that approximate reflections or take advantage of simplifying assumptions, such as having a planar reflector with straight edges.

11.1.1 Techniques for Rendering Reflections

It's easy and intuitive to imagine that a reflected scene is made up of "virtual" reflected objects. As shown in and Figure 55, the full scene can be thought of as a combination of the reflected scene painted "behind" the mirror object and the original scene.

The process of drawing a scene with a single reflector can be broken into two steps: creating virtual reflected objects from non-reflected objects, and then drawing the virtual objects within our original scene.

A reflected "virtual" object can be created by calculating a "virtual" vertex for every vertex in the original object. Several techniques of increasing sophistication for finding these vertices will be discussed, including the simple case of a planar reflector, curved surfaces with algebraic definitions such as spheres and cylinders, and arbitrary curved surfaces approximated by triangles. Beware that reflection will reverse vertex ordering for the faces in the object, so the face culling state for the reflection scene must be set to the reverse of the state for the original scene.

Once a reflected object's virtual vertices have been calculated, there are several techniques for rendering the reflected object within the scene. Clip planes may be used in restricted cases, the stencil buffer may be used as an arbitrary clipping region, or a texture map may be projected onto the reflector, using the reflector's edges to clip the reflected image. These techniques will be discussed in detail in the following sections.

11.1.2 Planar Reflectors

This section discusses the modeling of planar reflective surfaces. Three techniques are discussed: using the stencil buffer to draw the reflected geometry in the proper location, a technique using OpenGL's clip planes to clip reflected geometry to a reflected space visible by the eyepoint, and a technique which uses texture mapping to make an image of the reflected geometry which is then texture mapped onto the reflective polygon. Both techniques construct the scene in two (or more) passes.

As an example, consider a model of a room with a mirror on one wall. Compute the plane containing the mirror and define an eye point from which to render the scene. When drawing the reflected scene, first apply a transformation

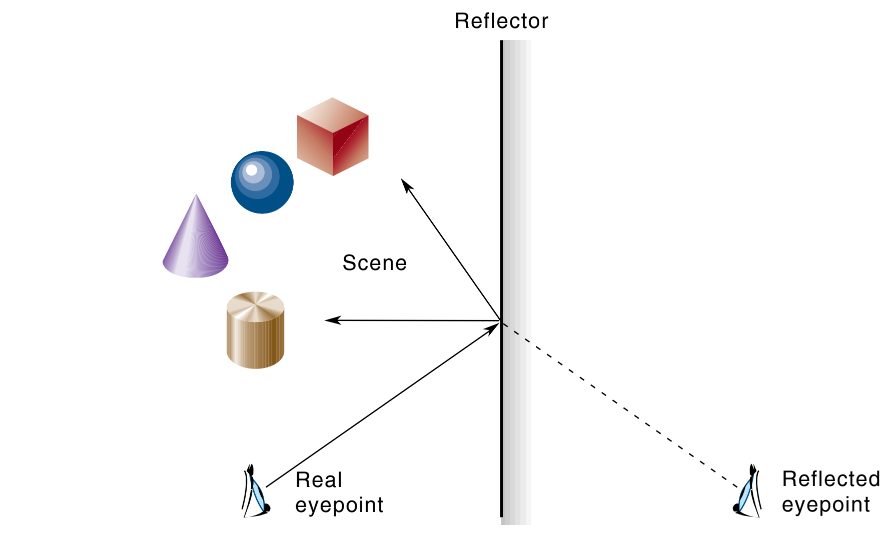


Figure 54. Mirror Reflection of the Viewpoint

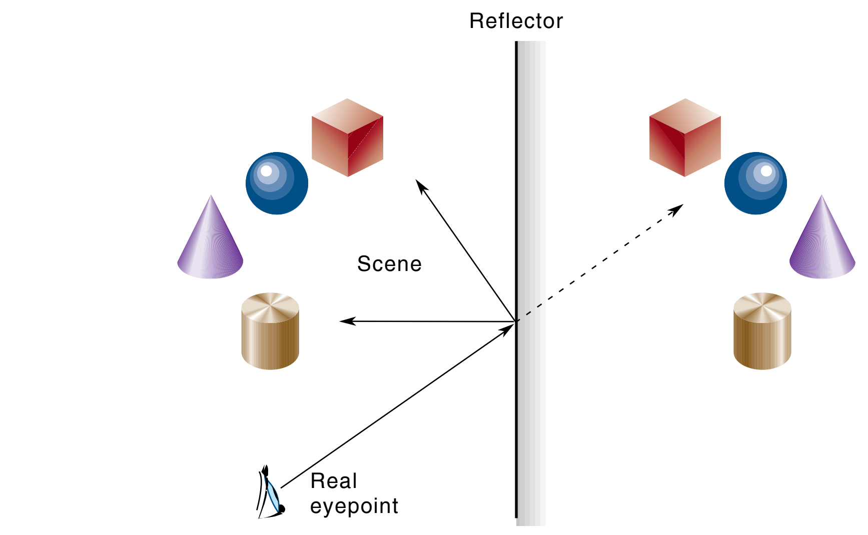


Figure 55. Mirror Reflection of the Scene

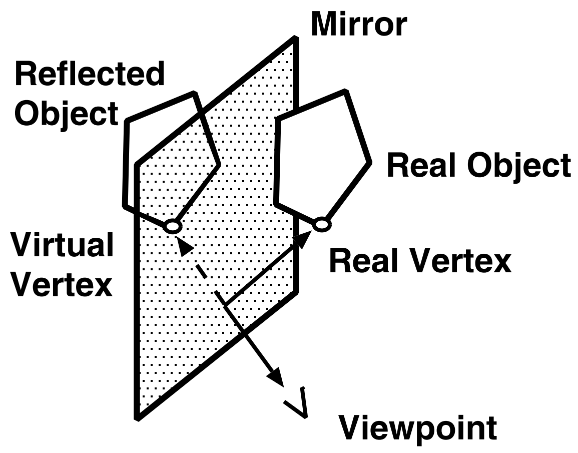


Figure 56. Virtual Reflected Vertices

that reflects objects across this plane. This can be envisioned as either reflecting the eye point or the objects across the plane; both are identical and some people find one approach more intuitive than the other.

The reflection transformation can be decomposed for convenience into a translation to the origin, a rotation mapping the mirror into the XY plane, a scale of -1 in Z, the inverse of the rotation previously used, and a translation back to the mirror location.

Given one vertex P of the planar reflector and a vector $\vec{V}$ perpendicular to the plane, this sequence of transformations can be expressed as the single 4 by 4 matrix R [34] shown below:

$$R = \begin{pmatrix} 1-2V_x^2 & -2V_xV_y & -2V_xV_z & 2(P\cdot V)V_x \\ -2V_xV_y & 1-2V_y^2 & -2V_yV_z & 2(P\cdot V)V_y \\ -2V_xV_z & -2V_yV_z & 1-2V_z^2 & 2(P\cdot V)V_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where

$$P \cdot V = P_xV_x + P_yV_y + P_zV_z$$

Applying this transformation to the original scene produces a virtual scene on the opposite side of the reflector representing the reflected scene. The next sections discuss how to render this reflected scene correctly within the non-reflected scene.

Planar Reflections Using the Stencil Buffer The effects of specular reflection can be approximated by a two-pass technique using the stencil buffer. The reflected scene is rendered during the first pass, and the non-reflected scene is rendered during the second pass, using the stencil buffer to clear parts of the reflected image outside the mirror polygon.

The sequence of steps for the first pass is as follows:

1. Set up initial viewing, and projection matrices as desired
2. Compute and apply the reflection transformation matrix.
3. Draw the remainder of the scene normally.

4. Undo the reflection transformation.

The application and removal of the mirror transformation can be easily applied as part of the modeling transformation stack, like OpenGL's `GL_MODELVIEW` matrix stack using `glPushMatrix` and `glPopMatrix`.

Objects drawn in the first pass look as they would when seen in the mirror, ignoring the fact that the mirror may not fill the entire field of view. For the purpose of rendering, imagine that the entire plane containing the mirror is reflective, but in reality the mirror does not cover the entire plane, and so parts of the scene may be drawn which will not be visible. For example, the lowest box in the scene in Figure 55 is drawn in the reflected scene, but its reflection is not actually visible in the mirror. The second pass draws over the parts of the reflection which are not normally visible.

When rendering the reflected scene, points on the plane of the reflector maintain the same position in eye space as when rendered without reflection. For example, the corners of the reflective polygon are in the same location when viewed from the reflected eye point as from the original viewpoint. In this way, it may be more intuitive to think of reflecting the scene, rather than the eye point.

It's important to configure a clipping plane in the plane of the mirror polygon, so geometry that is behind the mirror, and subsequently is reflected *in front* of the mirror, is not rendered.

Do not render the mirror during the first pass or it will obscure the reflected image. This problem is most easy solved by keeping track of the mirror polygon and skipping it during rendering. This should be simple because the algorithm already has to find the mirror object and its associated data.

A special effect created by a magnified or minified reflection can be implemented by moving the scene backwards or forwards along a vector perpendicular to the plane of the mirror. This scaling looks at first glance like a concave or convex mirror, respectively, but should not be expected to survive inspection. If the position is the same distance as the eye point from the mirror then the result is an image of the same scale.

Next, clear both the depth and stencil buffers, leaving the color buffer intact. Render the mirror polygon into the stencil buffer to mask out portions of the reflected scene which were drawn in the first pass and should remain visible. Configure the stencil test to pass *outside* the mirror polygon and clear the color buffer, so that pixels outside the mirror return to the background color. Finally, disable the stencil test and render the remainder of the scene normally.

The list of steps for the second pass, using OpenGL, is as follows.

1. Clear the stencil and depth buffers. (`glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`).
2. Configure the stencil buffer such that 1 will be stored at each pixel touched by a polygon:

```
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 1);
glEnable(GL_STENCIL_TEST);
```

3. Disable drawing into the color buffers (`glColorMask(0, 0, 0, 0)`).
4. Draw the mirror polygon, with blending if desired.
5. Reconfigure the stencil test:

```
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glStencilFunc(GL_NOTEQUAL);
```

6. Clear the color buffer to the background color.
7. Disable the stencil test (`glDisable(GL_STENCIL_TEST)`).
8. Draw the scene without reflection.

These steps are illustrated in Figure 57.

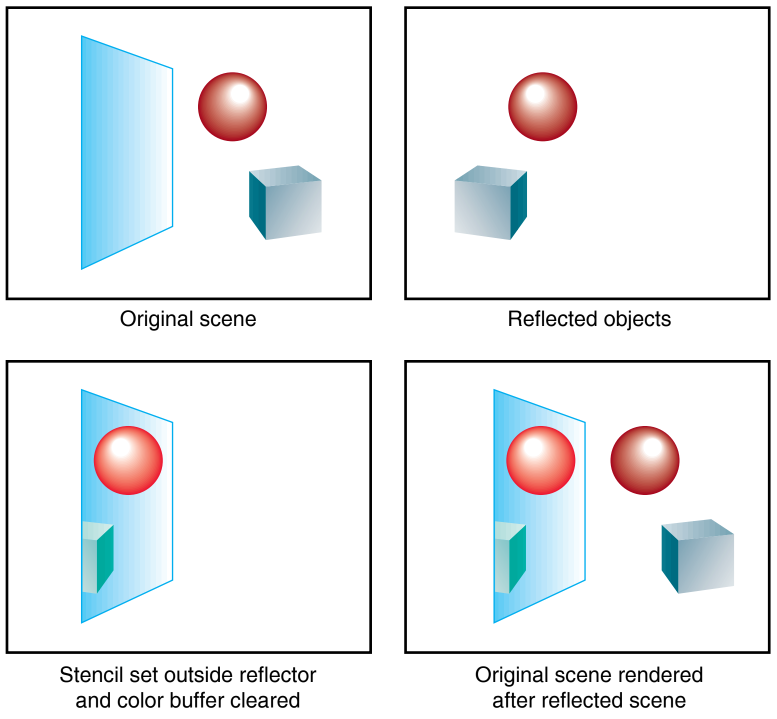

Original scene
Reflected objects
Stencil set outside reflector and color buffer cleared
Original scene rendered after reflected scene

Figure 57. Stencil Reflection Steps

It is possible to clear the stencil buffer and draw the reflector into stencil first, marking the stencil buffer where the reflection should be drawn. However, drawing the entire scene with stencil testing enabled is likely to result in lower performance than using stencil just to clear the screen all at once. A technique is presented later which requires setting stencil first to render interreflections.

The planar mirror reflector typically takes up a small region of the screen. Software that support culling to the viewing frustum can use a reduced frustum that tightly bounds the screen-space projection of the reflector when drawing the reflected scene, reducing the number of objects to be processed.

See Section 3 for more information on modeling.

Planar Reflections using Clip Planes Rather than use the stencil buffer to mask out the region outside the reflector, if the mirror polygon has five or fewer edges (four is a common number), OpenGL's clip planes can be configured to clip the reflected geometry to the region directly behind the mirror polygon.

For each edge, calculate the plane that is formed by that edge and the eyepoint. Configure this plane as a clip plane (without applying the reflection transformation). Finally, configure one plane to clip reflected polygons to the plane of the reflector, as in the stencil technique.

This may be a viable alternative if hardware does not support a stencil buffer.

Planar Reflections using Texture Mapping Another technique uses texture mapping. The first pass is identical to the first pass of the previous technique: draw the reflected scene. After drawing the scene, copy the image into

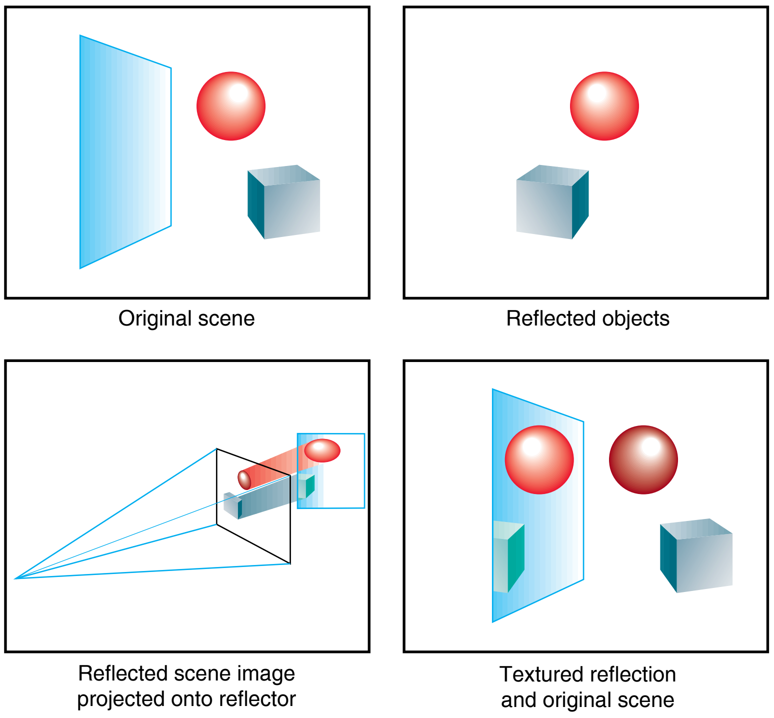

Original scene
Reflected objects
Reflected scene image projected onto reflector
Textured reflection and original scene

Figure 58. Masking Reflections Using Projective Texture

a texture. (For example, in OpenGL, use `glCopyTexImage2D`). During the second pass, map this texture onto the reflective polygon. The sequence of steps for the second pass is as follows:

1. Draw the non-reflected objects in the scene.
2. Bind the texture containing the reflected image.
3. Draw the reflective object with the appropriate texture coordinates.

The texture coordinates at the vertices of the reflective object must be in the same location as the vertices of the reflective object in the texture. These coordinates may be computed by figuring the projection of the corners of the object into the viewing plane used to compute the reflection map (the command `gluProject` may prove helpful).

Another approach, using OpenGL's texture coordinate generation and `GL_TEXTURE` matrix stack, is to load the texture matrix with the projection matrix and configure texture coordinate generation as `GL_EYE_LINEAR` with the `glTexGen` plane set to the XY plane. Texture coordinates are automatically generated that map the texture directly to pixels in the rendered image, as illustrated in Figure 58

The texture mapping technique may be more efficient on some systems than stencil buffering, but keep in mind that it uses up a texture stage, which may force the algorithm to use multiple passes to render the reflector if it also has a texture applied to it and the hardware supports only a single texture. On the other hand, you may be able to use a reflection texture during several frames, as described below.

The texture does not need to be the same size as the screen; it needs only to cover the screen-space projection of the reflector. If a smaller texture is used, however, the projected texture coordinates must be adjusted so the texture coordinates range from 0 to 1 within the smaller area.

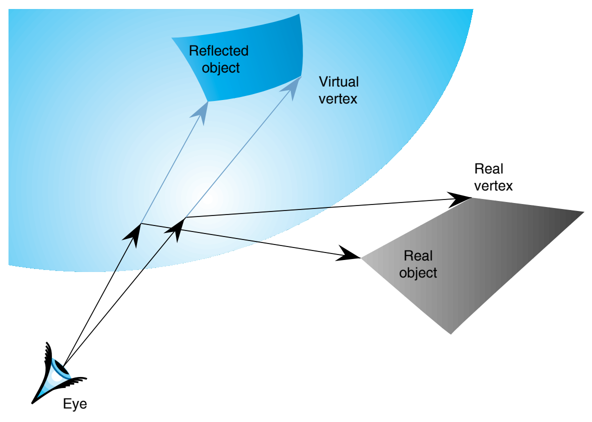


Figure 59. Virtual Reflected Vertices from a Curved Reflector

Finally, it may be acceptable to render the image of the reflected scene at a lower resolution than the final scene and let texture filtering blur the texture when it is projected onto the reflector.

11.1.3 Curved Reflectors

Implicit and Extruded Reflectors Since the objective with accelerated reflection is to find "virtual" vertices that appear to the viewer in the same place as an actual reflection, we can extend the planar technique to other reflection objects whose surface definitions are well-known.

For example, if our object is a sphere, the point on the sphere where a ray of light from vertex in our scene would reflect in order to reach the viewpoint can be computed directly [71]. Using the plane defined by that point and the normal to the sphere at that point, the real vertex can be reflected into a virtual vertex, as shown in Figure59 This reflection takes the place of the planar reflection transformation presented above, and is calculated for each reflected vertex rather than once per reflector.

The clip plane technique discussed for planar reflectors cannot be used directly for curved reflectors, although it may be acceptable to approximate the reflector surface for a single reflected face. The stencil masking and texture techniques, however, work well for this case.

Arbitrary Curved Reflectors Ofek and Rappoport present a technique [72] for computing virtual vertices for objects reflected in arbitrary curved reflectors represented by a mesh of triangles sharing per-vertex normals. The following sections examine their algorithm in the context of convex and then concave reflectors, and then discuss some caveats of the technique. Figure 60 is an example of a polygonal checkerboard reflected in a spherical patch formed from triangles.

Convex Reflectors Because there is no way to directly compute the reflection point from an arbitrary tessellated reflector, an acceleration data structure called an *explosion map* is used to approximate the reflection point.

An explosion map stores reflection directions mapped to a 2D image, in much the same way as OpenGL maps reflection directions to a sphere map, discussed in more detail below. A unit vector x, y, z is mapped into coordinates s, t within a circle inscribed in a explosion map with radius r using Equation 5 and 6.

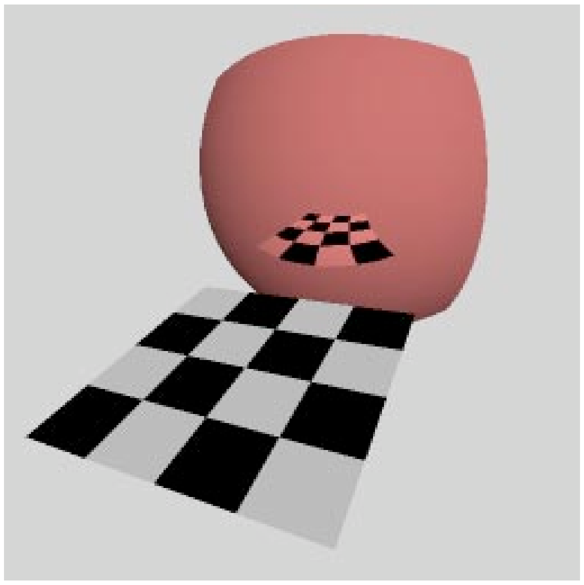


Figure 60. Checkerboard reflected in Curved Patch

$$s = \frac{r}{2}(1 + \frac{x}{\sqrt{2(z+1)}}) \tag{5}$$

$$t = \frac{r}{2}(1 + \frac{y}{\sqrt{2(z+1)}}) \tag{6}$$

The reflection directions used in the mapping are not the actual reflection rays determined from the reflector vertex and the viewpoint. Rather, the reflection ray is intersected with a sphere and the normalized vector from the center of the sphere to the intersection point is used instead. There is a one-to-one between reflection vectors from the convex reflector and intersection points on the sphere, as long as the sphere encloses the reflector. Figure 61 shows a viewing vector V and a point P on the reflector that forms the reflection vector R as a reflection of V. The normalized direction vector D from the center of a sphere to the intersection of R with that sphere is inserted into the equation shown above.

Once the reflection directions are mapped into 2D, an identifier for each triangle is rendered into the explosion map using the mapped vertices for that triangle. This provides an exact mapping from any point on the sphere to the point that reflects that point to the viewpoint. This identifier may be mapped using the color buffer or depth buffer or both as necessary. Applications will need to verify the resolution available in the frame buffer and will likely need to disable dithering. If the color buffer is used, the most significant bits of each component will be used as the least significant bits may not be stored at all and will be extended from the more significant bits on readback.

Imagine that a vertex of a face to be reflected lies on the sphere. (This is not typically the case, and will be addressed in the next paragraph.) For this vertex on the sphere, the explosion map can be used to find a reflection plane across which the vertex is reflected. The normalized vector pointing from the sphere center to the vertex is mapped into the explosion map to find a triangle ID. The mapped point formed from the vertex and the mapped triangle vertices are used to compute barycentric coordinates. These coordinates are used to interpolate a point and

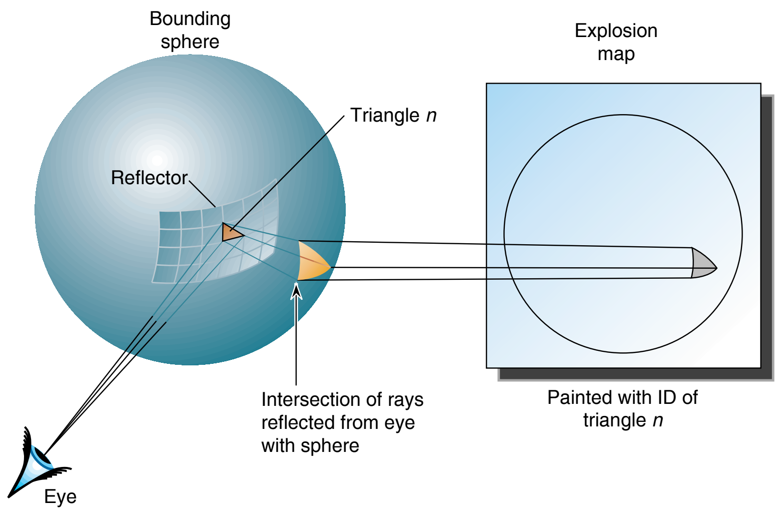

Bounding sphere
Explosion map
Triangle *n*
Reflector
Intersection of rays reflected from eye with sphere
Painted with ID of triangle *n*
Eye

Figure 61. Mapping Reflection Vectors into Explosion Map Coordinates

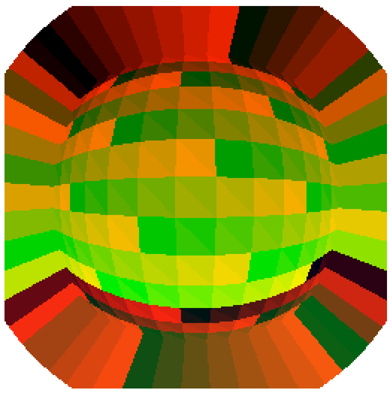


Figure 62. Triangle IDs Stored in an Explosion Map as Color

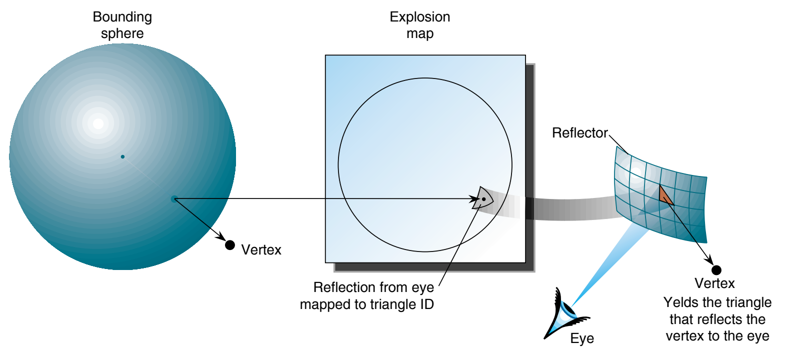

Bounding sphere
Explosion map
Reflector
Vertex
Reflection from eye mapped to triangle ID
Eye
Vertex
Yelds the triangle that reflects the vertex to the eye

Figure 63. Using An Explosion Map to determine the Reflecting Triangle

normal within in the triangle which approximate a plane and normal on the curved reflector. The mapped vertex is reflected across this plane to form a virtual vertex. This process is illustrated in Figure 63.

Since it is likely impossible to determine a sphere on which all vertices in the scene lie, two separate explosion maps with two spheres are computed. One sphere tightly bounds the reflector object, and one sphere bounds the entire scene. The normalized vector from the center of each sphere to the vertex is used to look up the reflecting triangle in the associated explosion map. Neither triangle may be correct, but the reflected virtual vertex is approximated by constructing virtual vertices using the results of each explosion map, and then interpolating between the two with a weight determined by the ratios of the distance from the surface of each sphere to the original vertex. Figure 64 shows how the virtual vertices determined from the explosion maps representing the near and far spheres are interpolated to find the final approximated reflected vertices.

Because the reflection directions from triangles in the reflector will not typically cover the entire explosion map, extension polygons will need to be constructed which extend the reflection mappings to cover the map. These extension polygons can be thought of as extending the edges of profile triangles in the reflector into quadrilaterals that fully partition space so that all vertices in the original scene are reflected by some polygon.

In the case that the reflector is modeled from a solid object, extension quadrilaterals may be formed from triangles in the reflector that have two vertex normals that face away from the viewer. Because the reflector is defined to be convex, these triangles automatically lie on the boundary of the front-facing triangles in the reflector. The normals of each vertex are projected into the plane perpendicular to the viewer at that vertex, which guarantees that the reflection vector from the normals maps into the explosion map. This profile triangle is projected into the explosion map using these “fixed up” coordinates. The edge formed by the “fixed up” vertices is extended to a quadrilateral to cover the remaining explosion map area, which is rendered into the explosion map with the profile triangle’s identifier. It is enough to extend these vertices just beyond the boundary of the explosion map before rendering this quadrilateral.

If the reflector is a surface and is not guaranteed to have back-facing polygons, it is necessary to extend the actual edges of the reflector until normals along the edge of the reflector fully span the space of angles in the X-Y plane.

Concave Reflectors In the case of reflectors formed from concave surfaces, the same techniques can be used with some observations. Vertices in the original scene may not map to exactly one reflection direction and thus their reflections cannot be approximated. Ofek and Rappoport note, however, that the motion of such vertices appears chaotic and any of the mapped virtual vertices can be chosen as the reflected vertex.

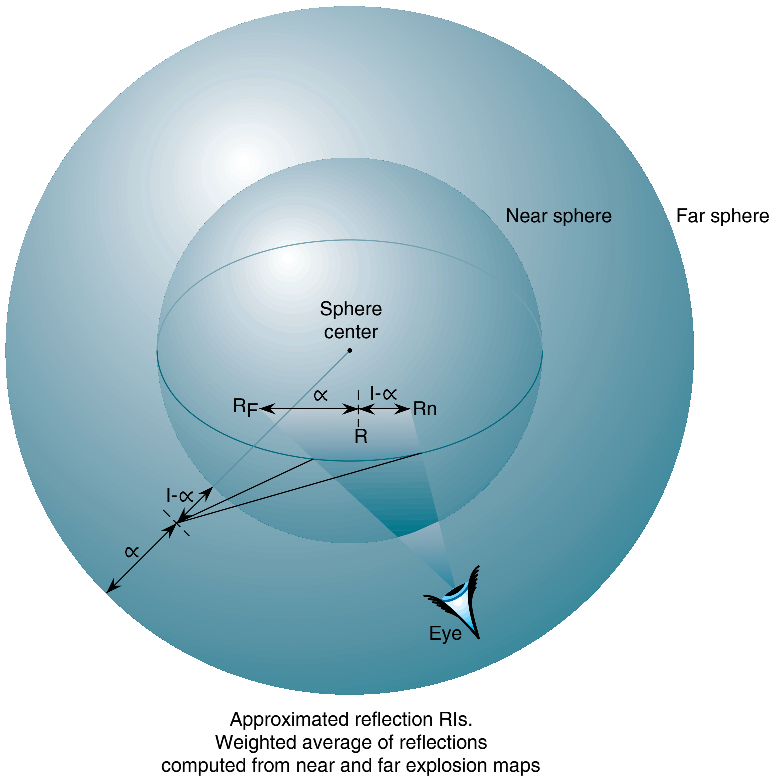

Near sphere
Far sphere
Sphere center
RF
∝
I-∝
Rn
R
I-∝
∝
Eye
Approximated reflection RIs.
Weighted average of reflections
computed from near and far explosion maps

Figure 64. Combining the Results of Near and Far Explosion Map Evaluation

Reflectors of Mixed Convexity Reflective surfaces may have to be decomposed into surfaces that are either concave or convex, and not both. It is enough, however, to decompose into reflectors whose reflection directions do not map into more than one location in the explosion map.

Conclusions and Issues Surfaces used in this algorithm must be tessellated sufficiently so that straight edges in the non-reflected scene form convincing curved reflections in the reflector. This may mean having to use some kind of hierarchical tessellation and error bounding or pre-tessellating the surface. The reflected objects have to be tessellated so that straight edges curve appropriately, but reflecting objects also must be tessellated so that the barycentric interpolation computed from the explosion map provides visually acceptable reflections.

On the other hand, because the existence of the reflected image may be enough to add the desired illusion of depth and location cuing, virtual objects typically can be rendered with less detail than the original objects, allowing both lower levels of detail and lower quality shading.

Both the texture mapping and stencil techniques discussed above can be used with this technique to mask away the portions of the reflected scene that are not actually visible.

11.1.4 Refraction

Refractions can be rendered with techniques similar to those presented for reflections. For example, with planar refractors, rather than reflecting the original scene about the mirror plane, translate and rotate the original scene to match the refracted viewpoint.

Refractions may be rendered using the same stencil masking and texture projection techniques described for reflections.

Refractions from a curved object can be rendered using an extension of the explosion map technique shown above, in which refraction directions are mapped to a 2D space instead of reflection directions.

Light rays converge through some curved refractors and diverge through others. Refractors which exhibit both behaviors must be partitioned into separate objects in the same way that concave and convex refractors are partitioned as discussed in Section 11.1.3.

11.1.5 Further Realism

Interreflections Either the stencil technique or the texture mapping technique may be used to model scenes with interreflections. Each algorithm uses additional passes for each “bounce” that the light takes. The application will need to determine the maximum number of interreflections to be rendered.

When using the stencil technique, it is necessary to rearrange the stencil operations so that the reflected scene images are masked directly by the stencil buffer. Render the reflections with the deepest recursion first. Concatenate the reflection transformations for each reflection polygon involved in an interreflection. Render each reflected image as follows:

1. Clear the stencil buffer.
2. Set the stencil operation to increment stencil values where pixels are rendered.
3. Render each reflector involved in the interreflection into the stencil buffer.
4. Set the stencil test to pass where the stencil value equals the number of reflections.
5. Apply planar reflection transformation overall, or apply curved reflection transformation per-vertex
6. Draw the reflected scene
7. Draw the reflector, blending as desired.

The choice of the initial color applied to reflectors in the scene can have an effect on the number of passes required. The initial reflection value will generally appear as a smaller part of the picture on each of the passes. A good initial guess is to set the initial color to the average color of the scene.

When using the texture technique, render with the deepest reflections first as above. The texture algorithm is more simple in that the only operations for each reflection are to apply the concatenated reflection transformations, copy the image to texture memory, and apply that image to the reflector during the next pass.

In an interactive application with moving objects or a moving viewpoint, it may be acceptable to use the reflection texture with the contents from the previous frame. This use of previous results is one of the advantages of the texture mapping technique.

Blurry Reflection Blurry reflectors, such as partially polished metals, can be approximated by accumulating several reflected images for which the reflected surface normal is perturbed according to the reflected ray distribution. This distribution can be sampled from a BRDF, for example.

For planar surfaces, the eyepoint is perturbed to simulate the perturbed normal. In this way, objects close to the mirror remain sharp while objects farther away are more blurred, which is what the viewer expects.

Similar techniques may be applied for curved reflectors and refractions.

Care must be taken to render enough samples to reduce visible error, otherwise reflected images tend to appear only like several images overlaid. The accumulation buffer may be used to average several reflection images before copying to texture in the texturing algorithm presented above.

11.2 Environment Mapping

The discussion of reflections so far is solidly based on the principles of geometric optics. Virtual reflected objects are rendered from virtual eye positions and then, through stenciling or reusing the virtual image as a texture, these virtual reflected objects are further transformed to appear as reflections on "real" reflective objects within the scene.

There is another approach however. Instead of rendering virtual versions of objects to appear as reflections on reflective surfaces, we can encode the complete panoramic environment surrounding a reflective object in a texture and then look up into the texture based on texture coordinates that are parameterized by the reflection vector as it varies across the reflective surface.

This technique is known as *environment mapping* and was first proposed by Blinn and Newell [11]. Unlike geometric reflection techniques that render reflected virtual objects to appear as reflections within the actual scene, an environment map may have nothing to do with the actual surroundings of the object. Indeed, a common use of environment mapping is interactively rendering an object free-floating against an empty black background. The point of applying an environment map to such an object is not to give a true sense of the object's surroundings (it has none). Instead, the view-dependent nature of the (fake) reflected environment merely helps highlight the object's surface curvature.

Additionally, the environment map image can be blurred to simulate the appearance of a dull, less shiny surface. An environment map may encode just the light from emissive light sources (instead of the less bright light reflected from non-emissive objects in the environment). This is the basis for the textured phong highlight technique described in Section 10.1.1.

At an abstract level, what an environment map supplies is a fast way to determine the incident irradiance in any direction at a particular fixed point in space. Given a direction, the environment map tells us the irradiance in that direction. Typically, the direction used for querying an environment is the reflection vector on a surface. Using a reflection vector to access an environment map accounts for the illumination due to perfect specular reflection [76].

An environment map is accessed *solely* based on a 3D orientation. This has important ramifications for the use of environment maps and is fundamental to most of the limitations of environment mapping. An orientation in 3D space ultimately has two degrees of freedom. For example, you can express a 3D orientation as some number of degree of latitude and and some other number of degrees of longitude. As will be seen, the actual parameterization of a 3D orientation into two degrees of freedom varies depending on the particular details of different environment mapping techniques.

The crucial observation is that given the two degrees of freedom used to define a 3D orientation, we can use 2D texture mapping as the basis for implementing environment mapping. By encoding an environment map as a 2D texture, OpenGL applications can use conventional 2D texturing hardware to accelerate environment mapping and render convincing reflective objects at very interactive rates.

The Inherent Limitations of Environment Mapping It is important to keep in mind the limitations of this approach. The environment map is accessed solely based on a 3D orientation. This would imply that different positions within the environment receive exactly identical incoming irradiance from the environment. Obviously, that's not at all true in general. The incident illumination for a point above a table is almost certainly going to be different than the incident illumination for a point underneath the table. Still the assumption is reasonable if the two positions are relatively near each other. What "near" means depends on the environment. In the case of the table, "near" certainly implies that the two points are close to each other but also that both should be on the same side of the table. However if the environment is the void of interstellar space, "near" could mean hundreds of miles away.

When using an environment map to render an object, we assume that the environment map is *relatively position independent* with respect to the object being rendering. This simply means, for example, that when we use a single environment map to render a shiny metallic teapot, we are assuming that the "environments" of all the points making up the teapot are all approximately the same. For convex objects such as a sphere that is relatively distant from the other objects making up the sphere's environment, this is a very reasonable assumption. For a non-convex object such as a shiny torus, the assumption is not very good because the torus, being non-convex, can reflect itself. Such inter-reflections are position dependent and therefore can not be captured by an environment map.

Even in a case such as a torus that has the opportunity for inter-reflections, we often still use environment mapping. The object's shiny surface simply will not reflect itself. Most observers will not even notice the lack of inter-reflections, particularly in the context of an animated scene. Environment mapping is often convincing even when applied in situations that, technically, are not well suited to the technique. The simple fact is that human observers are typically easily impressed by shiny rendered objects (they are a lot like raccoons in this respect) and are very trusting in the authenticity of reflections, particularly on complex curved surfaces.

Keep in mind that the environment of an object is a complete 360 degree panorama. This leads a problem when you try to encode the environment as a 2D texture. The implicit topology of OpenGL's 2D texturing functionality is that of an infinite plane with texture coordinates simply interpolated between vertices using (ideally) barycentric coordinates. Anyone versed in planar and spherical geometry understands the sort of problems this mismatch of topology creates.

Before we can use OpenGL texturing for environment mapping, we must first decide on a parameterization of the environment that works given the way OpenGL interpolates texture coordinates. Multiple parameterizations are possible. Blinn and Newell originally described a cylindrical parameterization indexed by polar coordinates. Unfortunately, 2D texture coordinates parameterized as an abscissa and polar angle are not interpolated correctly by conventional texturing hardware as supported by OpenGL. Figure 65 demonstrates this point.

Three common environment map parameterizations are sphere mapping, dual paraboloid mapping, and cube mapping. Each will be described in turn. Each parameterization has different advantages and disadvantages.

You can use environment maps to represent any effect that depends only upon a 3D orientation. These effects include specular and directional diffuse reflection, refraction, and Phong lighting. Several of these effects are

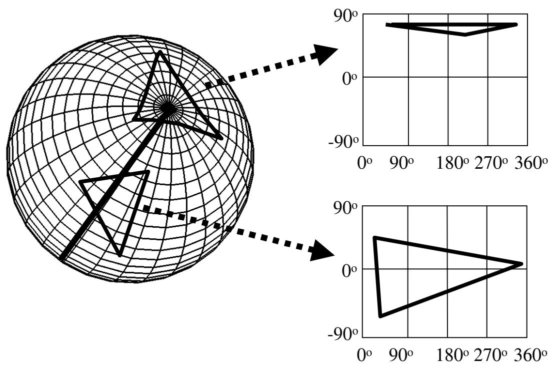


Figure 65. Problems with Cylindrical Environment Mapping. A triangle that covers a pole does not cover the pole when interpolated using polar coordinates and a cylindrical environment map encoded as a conventional 2D texture. Likewise, a triangle that crosses the line of zero degrees longitude does not cross the line as expected when 2D textured with OpenGL.

discussed in the context of OpenGL's sphere mapping capability, but they are applicable to other parameterizations as well.

11.2.1 Sphere Mapping

OpenGL has built in support for an environment mapping parameterization know as *sphere mapping*. Sphere mapping is a type of environment mapping in which the irradiance image is equivalent to that which would be seen in a perfectly reflective hemisphere when viewed using an orthographic projection [76]. This concept is illustrated in Figure 66. The sphere map is computed in the viewing plane. The width and height of the plane are equal to the diameter of the sphere. Rays fired using the orthographic projection are shown in blue (dark gray). In the center of the sphere, the ray reflects back to the viewer. Along the edges of the sphere, the rays are tangent and go behind the sphere.

Note that since the sphere map computes the irradiance at a single point, the sphere is thought of as infinitely small. In effect, you take the limit as the size of the sphere approaches zero. Note that a ray cast at the center of the sphere is bounced directly back at the viewer. Also note that all of the rays along the silhouette edge of the sphere will map to the same point directly behind the sphere in the environment.

OpenGL provides a texture coordinate generation mode to generate s and t texture coordinates at vertices based on the current normal and the direction to the eye point. The generated coordinates are then used to index a sphere map image which has been bound as a texture.

The Mathematics of Sphere Mapping The vector from the eye point to the vertex is denoted as $\vec{U}$, normalized to $\vec{U'}$. Since the computation is performed in eye coordinates, the eye is located at the origin and $\vec{U}$ is equal to the location of the vertex in eye-space. The current normal $\vec{N}$ is transformed to eye coordinates by the inverse transpose of the modelview matrix and is normalized, becoming $\vec{N'}$. The reflected vector $\vec{R}$ can be computed as:

$$\vec{R} = \vec{U'} - 2(\vec{N'} \cdot \vec{U'})\vec{N'} \tag{7}$$

Define:

$$p = \sqrt{R_x^2 + R_y^2 + (R_z + 1)^2}$$

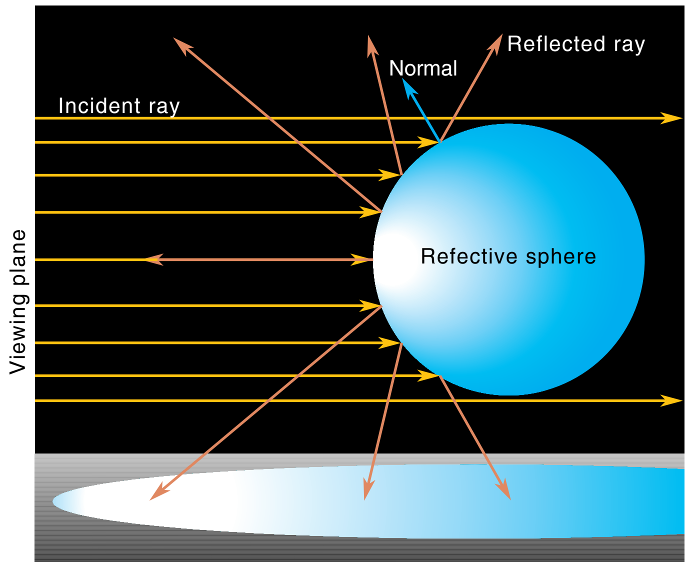


Figure 66. Creating a Sphere Map

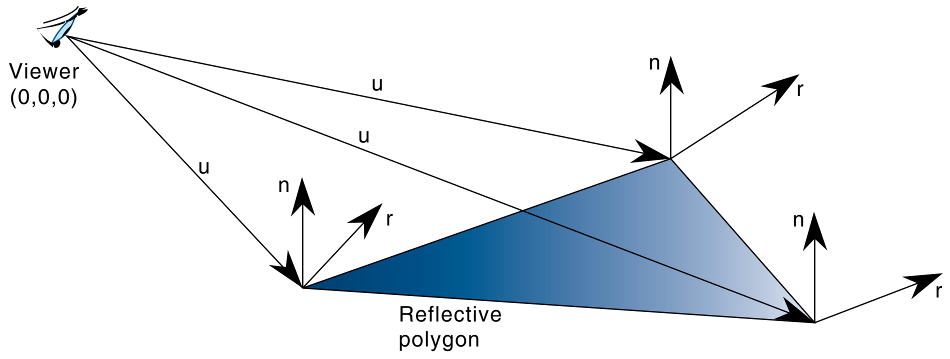


Figure 67. Sphere Map Coordinate Generation

Then the texture coordinates are calculated as:

$$s = \frac{R_x}{2p} + \frac{1}{2} \tag{8}$$

$$t = \frac{R_y}{2p} + \frac{1}{2} \tag{9}$$

This computation happens internally to OpenGL in the texture coordinate generation step. The math is performed for each vertex of a primitive as shown in Figure 67.

The above equations look intimidating, but they are not as mystifying as they first appear. Equation 7 is the standard equation for computing the reflection vector given a surface normal and incident vector. Most introductory graphics textbooks have a short proof and a good explanation of this equation.[9]

Because $\vec{N'}$ and $\vec{U'}$ are normalized, $\vec{R}$ is normalized as well. You can think of $\vec{R}$ as a 3D point on the unit sphere. Our task is to make this 3D point into a 2D texture coordinate. Reducing to two dimensions is accomplished by projecting $\vec{R}$ onto the unit circle in the $R_z = 0$ plane. This is the reason for dividing by p.

Because we know $\vec{R}$ is normalized, the result of R_x/p and R_y/p must be within the range $[-1, 1]$. But to index into a 2D texture map, we need coordinates in the $[0, 1]$ range. In Equations 8 and 9, the scale by $\frac{1}{2}$ and bias by $\frac{1}{2}$ serve to map values in the range the $[-1, 1]$ to the range $[0, 1]$.

When constructing sphere maps, the reverse mapping from (s, t) to $\vec{R}$ is often needed. The reverse mapping is:

$$\begin{aligned} R_x &= 2\sqrt{-4s^2 + 4s - 1 - 4t^2 + 4t}(2t - 1) \\ R_y &= 2\sqrt{-4s^2 + 4s - 1 - 4t^2 + 4t}(2s - 1) \\ R_z &= -8s^2 + 8s - 8t^2 + 8t - 3 \end{aligned}$$

The image in a sphere map can be thought of as a circle of radius $\frac{1}{2}$ centered at $(\frac{1}{2}, \frac{1}{2})$. Plugging $(\frac{1}{2}, \frac{1}{2})$ into the reverse mapping equations above results in a reflection vector of $(0, 0, 1)$. OpenGL assumes the eye looks down the negative Z axis. So the vector $(0, 0, 1)$ is reflected straight back at the eye. If we think of a sphere map as what we see when looking directly at a chrome sphere, then dead in the center of the sphere we expect to see our own reflection.

Try plugging (s, t) coordinates on the edge of the sphere map's circle such as $(0, \frac{1}{2})$ and $(\frac{1}{2}, 0)$ into the reverse mapping equations. For any point on the circle edge, the result is always $(0, 0, -1)$. Again, if we think of a sphere map as what we see when looking at a chrome sphere, the silhouette edge of the sphere is seen as an extreme grazing reflection of whatever is directly behind the sphere. If this does not make sense, look back at Figure 66 to convince yourself of this fact. This means that every point on the circle's edge of a properly generated sphere map should the same color.

Finally try plugging (s, t) coordinates for points outside the sphere map's circle into the reverse mapping equations. Try points such as $(0, 0)$ and $(1, 1)$. The operand of the square root in the equations for both R_x and R_y become negative. This should not be too surprising because properly normalized reflection vectors are guaranteed to fall within the sphere map's circle.

Using a Sphere Map To use sphere mapping in OpenGL, the following steps are performed:

1. Bind the texture containing the sphere map.
2. Set sphere mapping texture coordinate generation:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
```

[9]Many texts [27, 28, 82] present this reflection vector formula with a sign reversed from the form shown in Equation 7. It is the still the same fundamental formula; the difference is simply one of convention. The OpenGL $\vec{U}$ vector points from the eye to the surface vertex (i.e., an eye-space position), while many texts present the formula instead considering a light vector pointing from the surface to a light.

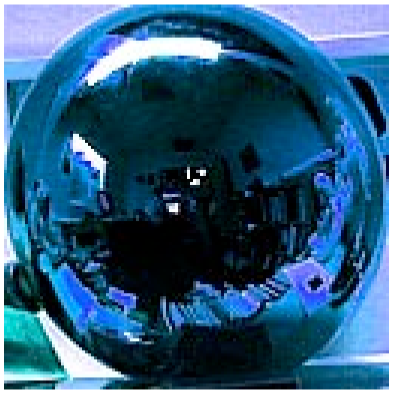


Figure 68. Reflection Map Created Using a Reflective Sphere

3. Enable texture coordinate generation and 2D texturing:

```
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_2D);
```

4. Draw the object, providing correct normals on a per-face or per-vertex basis.

Generating a Sphere Map for Specular Reflection There are several ways to generate a specular sphere map. Two physical approaches are worth mentioning. In the first approach, the user literally takes a picture of a reflective sphere. Figure 68 was generated in this fashion. This technique is problematic because the camera is visible in the sphere map. In the second approach, a fisheye lens approximates the sphere mapping. The problem with this technique is that no fisheye lens can provide the 360° field of view required for a correct result.

A sphere map can also be generated programmatically. Consider the circle of the environment map within the square texture to be a unit circle. For each point (s, t) in the unit circle, you can compute a point $\vec{P}$ on the sphere:

$$\begin{aligned} P_x &= s \\ P_y &= t \\ P_z &= \sqrt{1.0 - P_x^2 - P_y^2} \end{aligned}$$

Since you are dealing with a unit sphere, the normal at $\vec{P}$ is equal to $\vec{P}$. Given the vector $\vec{E}$ toward the eye point, you can compute the reflected vector $\vec{R}$:

$$\vec{R} = 2\vec{N}(\vec{N} \cdot \vec{E}) - \vec{E} \tag{10}$$

In OpenGL, it is assumed that the eye point is looking down the negative z axis, so $\vec{E} = (0, 0, 1)$. Equation 10 reduces to:

$$\begin{aligned} R_x &= 2N_xN_z \\ R_y &= 2N_yN_z \\ R_z &= 2N_zN_z - 1 \end{aligned}$$

The assumption that the $\vec{E} = (0, 0, 1)$ means that OpenGL's sphere mapping is actually not view-independent. The implications of this assumption will be discussed below with the other limitations of the sphere mapping technique.

The rays are intersected with the environment to determine the irradiance. A simple implementation of the algorithm is shown in the following pseudocode:

```
void gen_sphere_map(GLsizei width, GLsizei height, GLfloat pos[3],
                    GLfloat (*tex)[3])
{
  GLfloat ray[3], color[3], p[3], s, t;
  int i, j;

  for (j = 0; j < height; j++) {
    t = 2.0 * ((float)j / (float)(height-1) - .5);
    for (i = 0; i < width; i++) {
      s = 2.0 * ((float)i / (float)(width - 1) - .5);

      if (s*s + t*t > 1.0) continue;

      /* compute the point on the sphere (aka the normal) */
      p[0] = s;
      p[1] = t;
      p[2] = sqrt(1.0 - s*s - t*t);

      /* compute reflected ray */
      ray[0] = p[0] * p[2] * 2;
      ray[2] = p[1] * p[2] * 2;
      ray[3] = p[2] * p[2] * 2 - 1;
      fire_ray(pos, ray, tex[j*width + i]);
    }
  }
}
```

Note that you could easily optimize the routine such that the bounds on `i` in the inner `for` loop were intelligently set based on `j`.

The most interesting part of the computation has been encapsulated inside the `fire_ray` routine. `fire_ray` performs the ray/environment intersection given the starting point and the direction of the ray. Using the ray, it computes the color and puts the results into its third parameter (which is the appropriate location in the texture map).

A naive implementation such as the one above will lead to sampling artifacts. In reality, a texel in the image projects to a volume which should be intersected with the environment. To filter, you should choose several rays in this volume and combine the results.

The intersection and color computation can be done in several ways. You can use a model of the scene and a ray tracing package. Alternately, you can represent the scene as six images which form the faces of a cube centered around the point for which the sphere map is being created. The images represent what a camera with a 90° field of view and a focal point at the center of the square would see in the given direction. The six images may be generated with OpenGL or a rendering package, or can be captured with a camera. Figure 69 shows six images which were acquired using a camera. Once the six images have been acquired, the rays from the point are intersected with the cube to provide the sphere map texel values. Figure 70 shows the map generated from the cube faces in Figure 69.

Warping a Sphere Map from Cube Views Yet another approach uses OpenGL's texture mapping capabilities to create the sphere map. Unlike a ray casting approach, the texture warping approach to constructing sphere maps is easily accelerated by graphics hardware. The algorithm renders six cube faces as previously described and copies the results into six respective texture objects using `glCopyTexImage2D`. Be sure to align the first cube face view frustum to directly face the viewer. Then the other five cube faces should be aligned with respect to

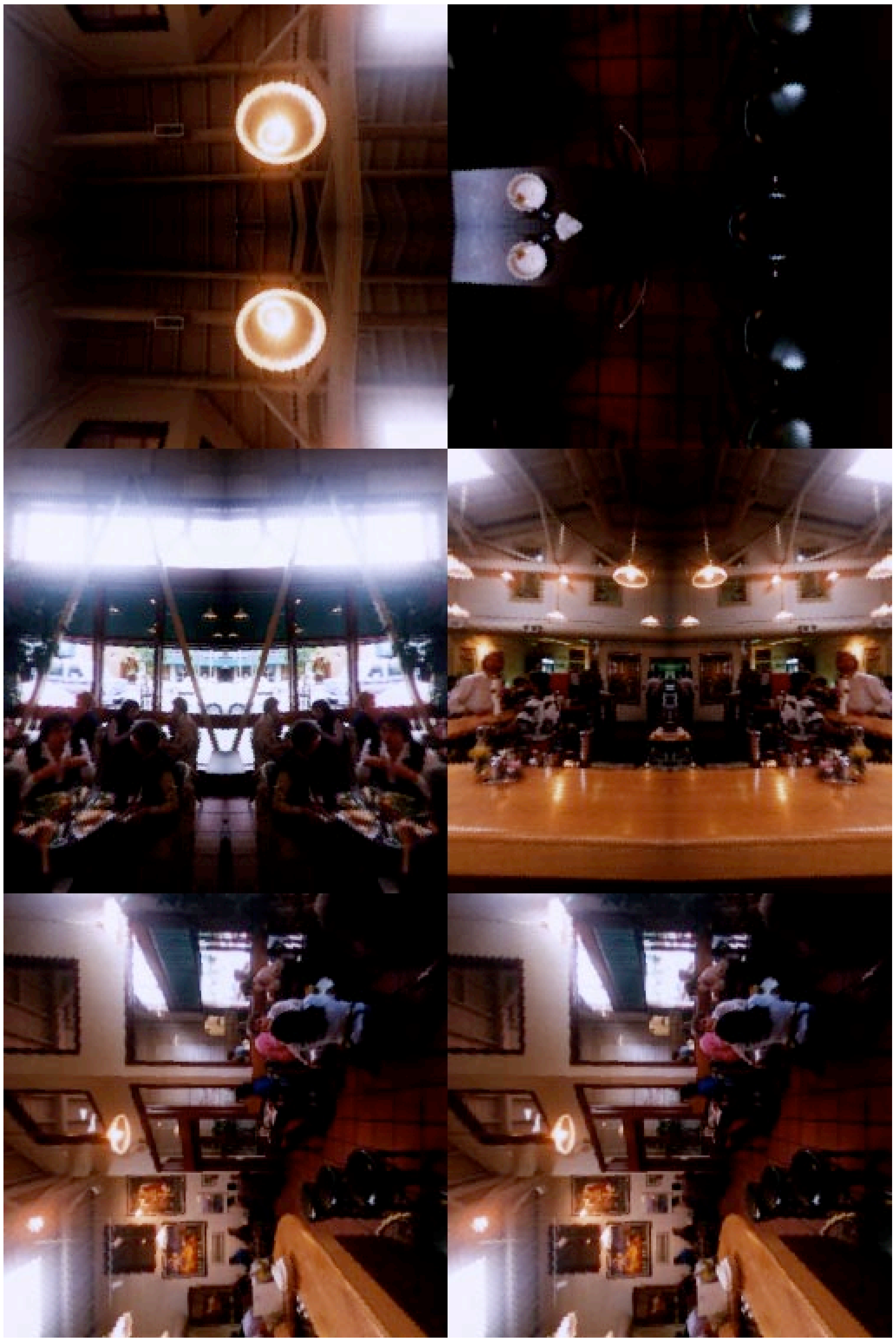


Figure 69. Image Cube Faces Captured at Cafe Verona in Palo Alto, California

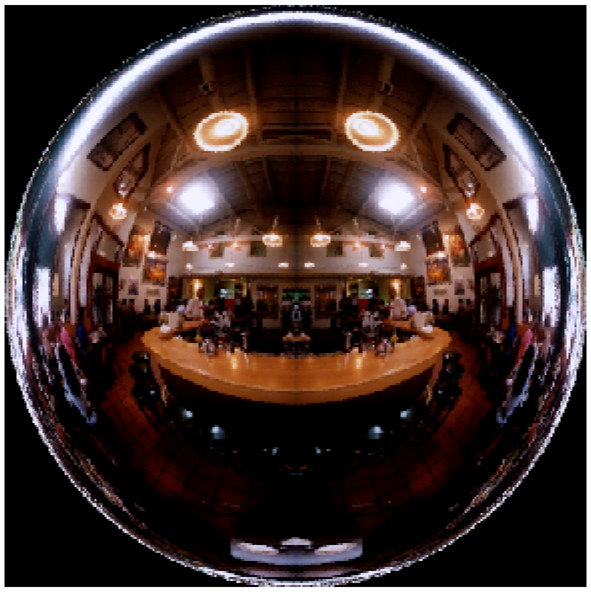


Figure 70. Sphere Map Generated from Image Cube Faces in Figure 69

the first. This reliance on the direction to the viewer is because a sphere map is *view dependent* which will be explained fully later.

Now, a textured mesh can be drawn to form the actual sphere map. Each one of the six faces becomes a subregion of the mesh. Figure 71 shows the relationship of the submesh for each cube face view to the entire sphere map mesh. The finer the tessellation of the mesh is, the better the warping. In practice, the mesh does not have to be very fine for constructing usable sphere maps.

To render the mesh, first map locations on the cube view faces to unnormalized reflections vectors. Think of each location on a cube view face as a 2D coordinate (u, v) where both u and v range between $[-1, 1]$. For example, consider a particular location $(0.3, -0.4)$ on the $Y = -1$ plane face (assuming the front cube view face is oriented on the $Z = +1$ plane so as to face the viewer). This location is mapped to the unnormalized reflection vector $(0.3, -1.0, -0.4)$. Normalize this vector and consider it $\vec{R'}$. Then with this $\vec{R'}$, compute (s, t) using Equations 8 and 9.

Treat (s, t) as a 2D *vertex position* in the range $[-1, 1]$ by $[-1, 1]$. Scale by one half and bias by one half the original 2D coordinate, the $(0.3, -0.4)$ coordinate in our example, to map it into the standard $[0, 1]$ by $[0, 1]$ texture image range. Use this remapped coordinate as a texture coordinate for the 2D vertex position. Setup an orthographic view mapping the $[-1, 1]$ by $[-1, 1]$ coordinate range into a 128 by 128 pixel region (or whatever resolution sphere map you want to create). Bind to the texture object containing the cube view face rendered image for appropriate face (the $Y = -1$ plane cube view face in our example). Then using the process just described for mapping from a cube face location to a reflection vector to a 2D coordinate, render a mesh of such coordinates, assigning them texture coordinates as described.

The back face mesh must be handled specially. It is not a straightforward warped rectangular patch, but instead is a mesh in the shape of a ring. You can think of the back cube view face as being pulled inside-out. In a sphere map, the center of the back cube view face becomes a singularity around the circular edge of the sphere map. The easiest way to render the back face mesh is as four meshes. The construction of these meshes is aided by the reverse mapping equations. Another issue with the back face mesh is that if a simple polygonal mesh is used, the

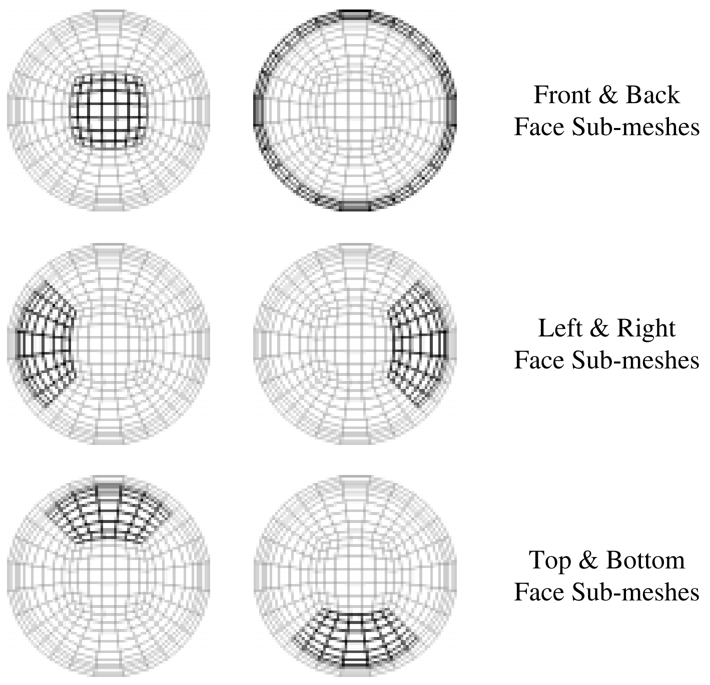


Figure 71. Meshes for Warping Six Cube Views into a Sphere Map

polygonal edges will not for a perfect sphere. For this reason, it is useful to add an narrow “extender” mesh at the circle’s edge that makes sure the entire circular sphere map region is rendered.

The meshes required are static for a fixed tessellation. You can compute the meshes once and then simply re-render the meshes for different sets of cube views. Precomputing the meshes helps reduce the overhead for repeated warping of cube face views into sphere map textures.

The final step is to copy the rendered sphere map into a texture using `glCopyTexImage2D`. Once this is done, you are ready to use the newly constructed sphere map.

Multipass Techniques and Interreflections Scenes containing two reflective objects may be rendered using sphere maps created via a multipass algorithm. Begin by creating an initial sphere map for each of the reflective objects in the scene. Choice of initial values was discussed in detail in Section 11.1.5. Then iterate over the objects, recreating the sphere maps with the current sphere maps of the other objects applied. The following pseudocode illustrates how this algorithm might be implemented:

```
do {
  for (each reflective object obj with center c) {
    initialize the viewpoint to look along the axis (0, 0, -1)
    translate the viewpoint to c
    render the view of the scene (except for obj)
    save rendered image as cube1
    rotate the viewer to look along (0, 0, 1)
    render the view of the scene
    save rendered image as cube2
    rotate the viewer to look along (0, -1, 0)
    render the view of the scene
    save rendered image as cube3
    rotate the viewer to look along (0, 1, 0)
    render the view of the scene
    save rendered image as cube4
    rotate the viewer to look along (-1, 0, 0)
    render the view of the scene
    save rendered image as cube5
    rotate the viewer to look along (1, 0, 0)
    render the view of the scene
    save rendered image as cube6
    using the cube images, update the sphere map of obj
  }
} while (sphere map has not converged)
```

Note that during the rendering of the scene, other reflective objects must have their most recent sphere maps applied. Detection of convergence can be tricky. The simplest technique is to iterate a certain number of times and assume the results will be good. More sophisticated approaches can look at the change in the sphere maps for a given pass, or compute the maximum possible change given the projected area of the reflective objects.

Other Sphere Mapping Techniques Sphere mapping may be used to approximate effects other the specular reflection. Any effect which is dependent only on the surface normal or other single vector can be approximated, including Phong shading and refractive effects. You can use your sphere map to store the outgoing color and intensity as a function of the normal. If you use a sphere map indexed by something other than the reflection vector, you will need to perform your own texture coordinate computations.

When computing your specular sphere map, this color was determined by firing a ray which had been reflected about the normal. To compute a different type of sphere map, determine the color using a different method. For

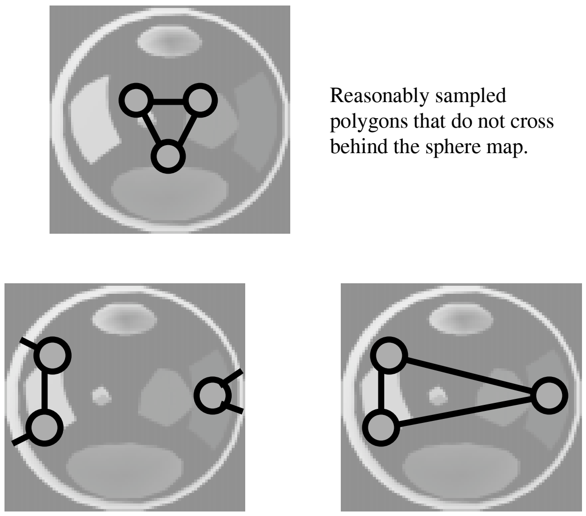

Reasonably sampled polygons that do not cross behind the sphere map.
Reasonable: Intended environment wrap through the sphere map perimeter.
Wrong: But 2D texturing hardware simply crosses the environment instead of wrapping.

Figure 72. The Source of Sphere Mapping Sparkles

example, to create a Phong lighting map, you can take the dot product of the normal direction and the direction to the light source.

Limitations of Sphere Mapping Although sphere mapping is generally convincing, it is not generally correct. Most of the artifacts come from the fact that the sphere map is generated at a single point and then applied over a large number of points. Objects with interreflections cannot be handled correctly. If reflected objects are close to the reflective object, their reflections should appear differently when viewed from different points on the reflector. Using sphere maps, this will not happen. Sphere mapping results are only correct if you assume that all the reflective objects are infinitely far from the reflective object.

Interpolation of the texture coordinates also leads to artifacts. Texture coordinates are computed at the vertices and linearly interpolated across the polygon. Unfortunately, the sphere map is not in a linear space, so this interpolation is not correct. Additionally, the linear interpolation will not take into account the fact that the points at the edge of the circle all map to the same location. Coordinates may be interpolated within the circle of the sphere map when they should be interpolated across the boundary [98]. Figure 72 shows how this occurs.

This is responsible for an unsightly artifact that appears as random "sparkles" or "dirt" at the silhouette edge of a sphere mapped object. As long as the sphere map center reflects directly back at the viewer and reasonable vertex normals are supplied, the artifacts are only a problem at the silhouette or grazing edges of objects because that is where vertex normals of a polygon are likely to cause wrapping to the other side of the sphere map. Because these grazing polygons are small in screen space, the number of affected pixels is usually small. Still the effectively random sparkling can be objectionable, particularly in animated scenes. Figure 73 shows a scene and a zoomed version of the scene showing sparkles at the silhouette edge of the sphere mapped object.

The final major limitation of sphere maps is that their construction assumes that the center of the sphere map is what reflects directly back at the viewer. Recall that in the discussion about the construction of sphere maps, both by ray casting or by texture warping, the construction involves a particular view orientation. The sphere map

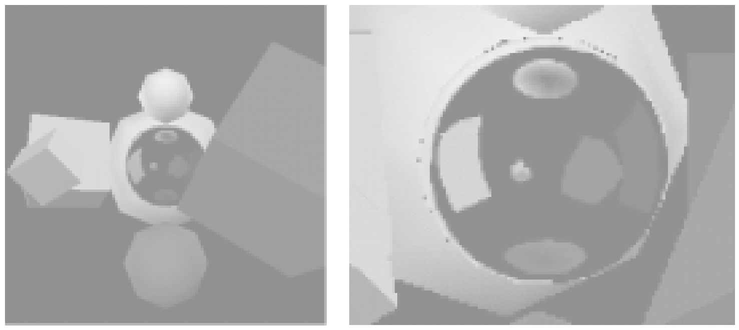


Figure 73. Example Showing Sparkle Artifacts

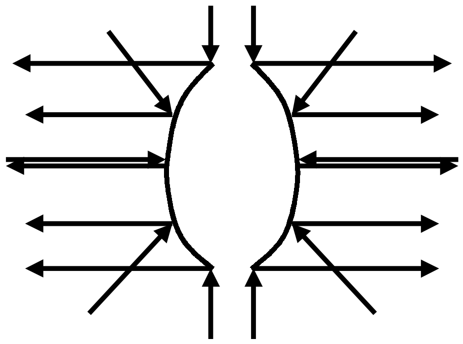


Figure 74. Two Paraboliods Shown in 2D as Parabolas

image is said to be *view dependent*. This means unless the sphere map is regenerated for different views, the sphere map will be incorrect.

It is possible using the six cube map views to continuously re-warp a new sphere map texture for different views. If the environment mapped object is to reflect a dynamic environment, this continuous re-warping is required anyway. This ends up being a major limitation for using sphere map in complex dynamic scenes with a continuously changing viewer.

11.2.2 Dual-Paraboloid Environment Mapping

Another environment map parameterization proposed by Heidrich and Seidel [51] avoids many of the disadvantages of and objectionable artifacts caused by sphere mapping. The dual-paraboloid environment mapping approach is view independent, has better sampling characteristics, and, because the singularity at the edge of the sphere map is eliminated, there are no sparkling artifacts at glancing edges.

The Mathematics of Dual-Paraboloid Maps The principle that underlies paraboloid maps is the same principle that underlies a parabolic lens or satellite dish. The geometry of a paraboloid can focus rays. The paraboloid used

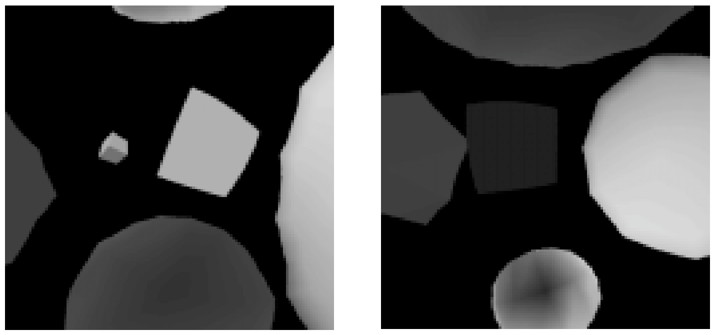


Figure 75. Example Dual-Paraboloid Texture Map Images

for dual-paraboloid mapping is:

$$f(x,y) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2), \qquad\qquad x^2 + y^2 \leq 1$$

Figure 74 shows how two paraboloids can focus the entire environment surrounding a point into two images.

Unlike the sphere mapping approach that the encodes the entire environment in a single texture, the dual-paraboloid mapping scheme requires two textures to store the environment, one texture for the "front" environment and another texture for the "back" environment. Importantly, the sense of "front" and "back" is completely independent of the viewer orientation. Figure 75 shows an example of two paraboloid maps. Because two textures are required, the technique must be performed in two rendering passes though this can be reduced to a single rendering pass if multitexturing is supported.

Because the math for the paraboloid is all linear (unlike the spherical basis of the sphere map), Heidrich and Seidel observe that OpenGL with its texture matrix can map a eye-coordinate reflection vector $\vec{R}$ into a 2D texture coordinate (s,t) within a dual-paraboloid map. Construct the necessary texture matrix as follows:

$$\begin{pmatrix} s \\ t \\ 1 \\ 1 \end{pmatrix} = \mathbf{A} \cdot \mathbf{P} \cdot \mathbf{S} \cdot (\mathbf{M}_l)^{-1} \cdot \begin{pmatrix} R_x \\ R_y \\ R_z \\ 1 \end{pmatrix}$$

where

$$\mathbf{A} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

is a matrix that scales an biases a 2D coordinate in the range $[-1,1]$ to the texture image range $[0,1]$. And where

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

is a projective transform that divides by the z coordinate. This serves to flatten a 3D vector into 2D. And where

$$\mathbf{S} = \begin{pmatrix} -1 & 0 & 0 & d_x \\ 0 & -1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

is a matrix that subtracts the supplied 3D vector from an orientation vector $\vec{d}$ that supplies a view direction. We will make $\vec{d}$ either $(0,0,-1)^T$ or $(0,0,1)^T$ depending on whether we are mapping the front or back paraboloid map respectively. Finally, the matrix $(\mathbf{M}_l)^{-1}$ is the inverse of the linear part of the current (affine) modelview matrix. The matrix $(\mathbf{M}_l)^{-1}$ transforms a 3D eye-space reflection vector into an object-space version of the vector.

Using Dual-Paraboloid Maps For the rationale for these transformations, consult Heidrich and Siedel [51]. The implication of this math is that these successive transformations can be concatenated into a single 4 by 4 projective matrix and then installed as OpenGL's texture matrix. Then supplying a per-vertex eye-space reflection normal via `glTexCoord3f`, the 3D vector will be transformed into a 2D texture coordinate in a front or back paraboloid map, depending on how $\vec{d}$ is oriented.

An alternative to supplying the reflection vector through `glTexCoord3f` is using the `NV_reflection_vector` extension's capability to generate automatically the eye-space reflection vector for the (s,t,r) texture coordinates. Recall that this is the same extension used in Section 6.15 though we are generating the reflection vector instead of the normal vector. Assuming the extension is available, here is how to generate the eye-space reflection vector as a 3D texture coordinate:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_NV);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_NV);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_NV);

glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
```

Here is how to set up OpenGL's texture matrix given the transformation described above:

```
GLfloat mapMatrix[16] = {
  0.5, 0,   0,  0,
  0,   0.5, 0,  0,
  0,   0,   1,  0,
  0.5, 0.5, 0,  1
};
GLfloat projectMatrix[16] = {
  1,  0,  0,  0,
  0,  1,  0,  0,
  0,  0,  1,  1,
  0,  0,  0,  0
};
GLfloat diffFrontMatrix[16] = {
  -1,  0,  0,  0,
   0, -1,  0,  0,
   0,  0,  1,  0,
   0,  0, -1,  1
};
GLfloat diffBackMatrix[16] = {
  1,   0,  0,  0,
  0,  -1,  0,  0,
  0,   0,  1,  0,
  0,   0,  1,  1
};
GLfloat mv[16], ilmv[16];

glGetFloatv(GL_MODELVIEW_MATRIX, mv);
mv[3]  = 0;
```

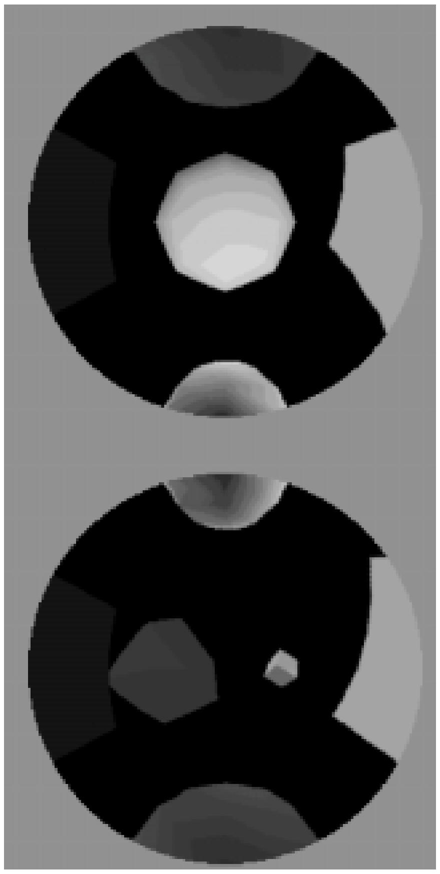


Figure 76. The Sweet Circles of a Dual-Paraboloid Map

```
mv[7]  = 0;
mv[11] = 0;
mv[12] = 0;
mv[13] = 0;
mv[14] = 0;
mv[15] = 1;
invert_matrix(mv, ilmv);

glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glMultMatrixf(mapMatrix);
glMultMatrixf(projectMatrix);
if (frontSide)
  glMultMatrixf(diffFrontMatrix);
else
  glMultMatrixf(diffBackMatrix);
glMultMatrixf(ilmv);
```

Note that each dual-paraboloid texture contains an incomplete version of the complete environment. There is some overlap between the two texture maps as shown in Figure 75 in the corners of each image. Notice that the corner region in one map are distorted so that the other map has a better sampled version of the same information. Moreover, there is some information in each map that is simply not in the other map. The information in the center of each map is only in a single map. Figure 76 shows that each map has a centered circular region where texels within the region are better sampled than the corresponding texels for the same reflection vector in the other map if the corresponding texels are available in the other map at all. We call this centered circular region of each dual-paraboloid map the *sweet circle*.

What remains to be done is making sure that the front dual-paraboloid map is used for pixels best sampled from the front dual-paraboloid map and vice versa. In the projective transformation discussed above, if a reflection vector falls within the sweet circle of one dual-paraboloid map, it will be guaranteed to fall outside the sweet circle of the opposite map.

With OpenGL's alpha testing capability, we can discard texels outside the sweet circle of each texture. The idea is to encode in the alpha channel of each dual-paraboloid texture an alpha value of 1.0 if the texel is within the sweet circle and 0.0 if the texel is outside the sweet circle. Be conservative about whether a texel is inside the circle to avoid in cracks the transition between the two maps.

Now, we can render an object in two passes. First, bind to the front dual-paraboloid texture and use a $\vec{d}$ value of $(0, 0, -1)$ when constructing the texture matrix. Then in a second pass, bind to the back dual-paraboloid texture and use a $\vec{d}$ value of $(0, 0, 1)$ when constructing the texture matrix. During both passes, enable alpha testing to eliminate fragments with an alpha value less than 1.0. Set up the texture environment to make the fragment's alpha value be the texture's alpha value. The result is a complete dual-paraboloid mapped object.

When multitexturing is available, the two passes can be collapsed into a single multitextured rendering pass. As described in Section 6.2, each texture unit has its independent texture matrix. We load the first texture unit to use the front texture matrix and the second texture unit to use the back texture matrix. The first texture unit uses a `GL_REPLACE` texture environment while the second texture unit uses a `GL_BLEND` texture environment. This effect is to blend the two textures based on the alpha component of the second texture. One side benefit of the multitextured approach is that the transition between the two dual-paraboloid map textures is harder to notice. Even with simple alpha testing the seam is quite difficult to notice.

Advantages and Disadvantages The main advantages of the dual-paraboloid map approach compared to the sphere map approach are better sampling of the texture environment, the elimination of sphere mapping's sparkle artifacts, and view-independence. The last advantage is important because it allows the viewer, environment mapped object, and the environment to move with respect to each other without having to continuously regenerate the dual-paraboloid map. The disadvantage of the dual-paraboloid map approach are that it requires two rendering passes or the use of multitexturing. Also constructing the dual-paraboloid map requires warping two textures instead of just one.

Even though they are view-independent, dynamic generation of dual-paraboloid maps is still necessary if you want the environment to be dynamic. The same texture warping approach that is used to construct sphere maps can be applied to generate dual-paraboloid maps though the mesh used is different. Figure 77 shows how cube map faces are arranged within the two dual-paraboloid map texture images, and Figure 78 shows what the texture warping mesh pattern looks like.

To help in the construction of the texture warping mesh, the dual-paraboloid mapping functions for converting a reflection vector $\vec{R}$ to the front and back 2D texture coordinates (s, t) are:

front side:

$$s = \frac{R_x}{1 - R_z}$$

$$t = \frac{R_y}{1 - R_z}$$

back side:

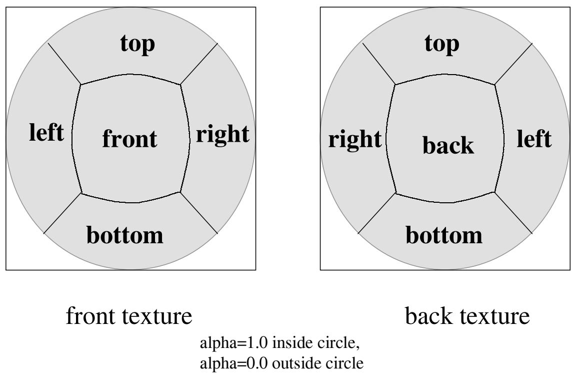


Figure 77. How Cube Map Faces Map to a Dual-Paraboloid Map

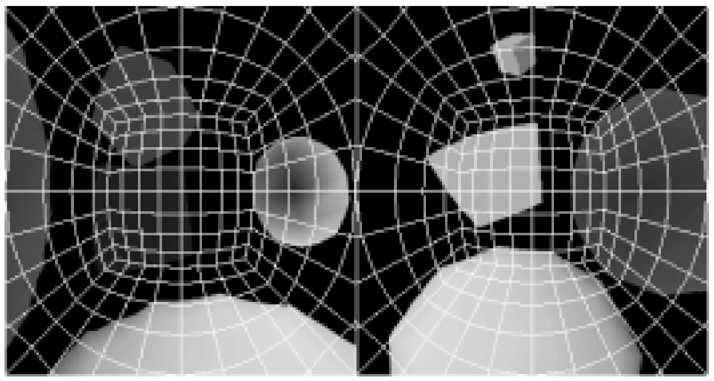


Figure 78. The Texture Warping Mesh for Constructing a Dual-Paraboloid Map

$$
\begin{aligned}
s &= -\frac{R_x}{1+R_z} \\
t &= -\frac{R_y}{1+R_z}
\end{aligned}
$$

The reverse mapping is:

front side:

$$
\begin{aligned}
R_x &= \frac{2s}{s^2+t^2+1} \\
R_y &= \frac{2t}{s^2+t^2+1} \\
R_z &= \frac{-1+s^2+t^2}{s^2+t^2+1}
\end{aligned}
$$

back side:

$$
\begin{aligned}
R_x &= \frac{-2s}{s^2+t^2+1} \\
R_y &= \frac{-2t}{s^2+t^2+1} \\
R_z &= \frac{1-s^2-t^2}{s^2+t^2+1}
\end{aligned}
$$

Cheap Per-pixel Lighting One application of dual-paraboloid texture mapping deserving particular note is implementing a view-independent form of textured per-pixel lighting. Two textures can encode a specular lighting solution as a dual-paraboloid map. Another two textures can encode a diffuse lighting solution as a dual-paraboloid map. By indexing the specular dual-paraboloid map by the eye-space reflection vector and the diffuse dual-paraboloid map by the eye-space normal vector, a complete specular-diffuse lighting solution is possible.

The same approach could be used with sphere mapping, but the cost of regenerating and downloading two sphere maps for the diffuse and specular contributions whenever the view changes will undermine good performance. Because dual-paraboloid maps are view-independent, the same dual-paraboloid maps can be used in changing views. Additionally, the sphere map sparkling artifacts are not an issue when using dual-paraboloid maps.

You can support an unlimited number of lights through this approach with no extra cost beyond that required to construct the dual-paraboloid maps. Directional lights are easily supported, but supporting positional requires constructing distinct dual-paraboloid maps for localized regions of space and assuming such regions are relatively positional independent with respect to lighting. Essentially, positional lights must be treated as directional lights for a localized region where the dual-paraboloid maps are used. High-quality shiny specular contributions may require increasing the resolution in the specular dual-paraboloid texture map. Modulating the texture color with the interpolated per-vertex color provides the equivalent of fast per-vertex color material changes. Spotlights and attenuation are not possible with this approach. Because the texture look ups into each dual-paraboloid map are performed per-pixel, this approach is rightfully considered a per-pixel lighting method. Unlike per-vertex lighting models, this approach can reproduce consistent specular highlights even on relatively poorly tessellated geometry.

With a single texture unit, this approach requires four textured rendering passes. After two passes to generate the diffuse lighting contribution, use additive blending to add in the specular contribution in the third and fourth specular passes. Modulating the diffuse lighting contribution with a surface texture requires an initial fifth rendering pass. Existing multitexture hardware supporting two texture units can implement this technique in two passes and add a surface texture with a third pass. Future hardware supporting five texture units and a suitably extended texture environment to combine all these units (the `ARB_multitexture` base texture environment is not capable enough) could render the entire per-pixel lighting effect including a surface texture in a single pass!

The `NV_reflection_vector` extension provides texture coordinate generation modes for both the eye-space reflection vector and eye-space normal vector to support this approach. Interpolating an unnormalized reflection vector and normal vector is not ideal. In true per-pixel Phong shading [78], the eye and normal vectors are interpolated and re-normalized per-pixel and the reflection vector is computed per-pixel.

11.2.3 Cube Environment Mapping

A third parameterization of environment mapping uses six cube face views directly as an environment map instead of requiring a re-warping of the cube views into a sphere map or dual-paraboloid map. To support cube mapping, the OpenGL implementation's texturing hardware is expected to directly fetch texels from the six cube face views loaded into texture memory. Existing OpenGL implementations do not support such a mode.

This approach is described by Voorhies and Foran [98]. Voorhies and Foran make two important observations. First, the expensive divider required in perspective-correct texture mapping hardware can be used by cube map texturing hardware to pre-construct the per-fragment divide necessary to project an unnormalized reflection vector to a particular cube face. Second, an efficient hardware block is proposed to compute the reflection vector per-pixel using linearly interpolated eye-space normal and eye-space position vectors. Unfortunately, efficient cube mapping requires special hardware support that is not available at the time of this writing.

11.3 Impact of Complexity on Choice of Reflection Technique

Because maintainability and readability of applications is important, it may be worth considering more general techniques like ray casting and ray tracing for providing reflections and shadows. As multiprocessor machines slowly become more prevalent, it may be the case that a simple brute-force algorithm provides acceptable performance without the complexity of the combination of the above techniques.

For example, an application implementing the algorithms above for curved reflectors and blurred reflections including view-frustum culling and reducing the size of textures used approaches a prohibitive complexity level. For small reflectors, ray tracing can achieve interactive performance with much less algorithmic complexity.

Unfortunately, there is no right answer and each developer must evaluate their tolerance for complexity, frame rate, and quality.

11.4 Creating Shadows

Shadows are an important way to add realism to a scene. There are a number of trade-offs possible when rendering a scene with shadows [104]. Just as with lighting, there are increasing levels of realism possible, paid for with decreasing levels of rendering performance.

Shadows are composed of two parts, the umbra and the penumbra. The umbra is the area of a shadowed object that is not visible from any part of the light source. The penumbra is the area of a shadowed object that can receive some, but not all of the light. A point source light would have no penumbra, since no part of a shadowed object can receive part of the light.

Penumbras form a transition region between the umbra and the lighted parts of the object; they vary as function of the geometry of the light source and the shadowing object. Since shadows tend to have high contrast edges, They are more unforgiving with respect to aliasing artifacts and other rendering errors.

Although OpenGL does not support shadows directly, there are a number of ways to implement them with the library. They vary in difficulty to implement, and quality of results. The quality varies as a function of two parameters. The complexity of the shadowing object, and the complexity of the scene that is being shadowed.

11.4.1 Projection Shadows

An easy-to-implement type of shadow can be created using projection transforms [96, 10]. An object is simply projected onto a plane, then rendered as a separate primitive. Computing the shadow involves applying a orthographic or perspective projection matrix to the modelview transform, then rendering the projected object in the desired shadow color.

Here is the sequence needed to render an object that has a shadow cast from a directional light on the z axis down onto the x, y plane:

1. Render the scene, including the shadowing object in the usual way.
2. Set the modelview matrix to identity, then call `glScalef(1.f, 0.f, 1.f)`.
3. Make the rest of the transformation calls necessary to position and orient the shadowing object.
4. Set the OpenGL state necessary to create the correct shadow color.
5. Render the shadowing object.

In the last step, the second time the object is rendered, the transform flattens it into the object's shadow. This simple example can be expanded by applying additional transforms before the `glScalef` call to position the shadow onto the appropriate flat object. Applying this shadow is similar to decaling a polygon with another coplanar one. Depth buffering aliasing must be taken into account. To avoid depth aliasing problems, the shadow can be slightly offset from the base polygon using polygon offset, the depth test can be disabled, or the stencil buffer can be used to ensure correct shadow decaling. The best approach is probably depth buffering with polygon offset. This way the depth buffering will minimize the amount of clipping you will have to do to the shadow.

The direction of the light source can be altered by applying a shear transform after the `glScalef` call. This technique is not limited to directional light sources. A point source can be represented by adding a perspective transform to the sequence.

Although you can construct an arbitrary shadow from a sequence of transforms, it might be easier to just construct a projection matrix directly. The function below takes an arbitrary plane, defined as a plane equation in $Ax + By + Cz + D = 0$ form, and a light position in homogeneous coordinates. If the light is directional, the w value should be 0. The function concatenates the shadow matrix with the current matrix.

```
static void
myShadowMatrix(float ground[4], float light[4])
{
    float  dot;
    float  shadowMat[4][4];

    dot = ground[0] * light[0] +
          ground[1] * light[1] +
          ground[2] * light[2] +
          ground[3] * light[3];

    shadowMat[0][0] = dot - light[0] * ground[0];
    shadowMat[1][0] = 0.0 - light[0] * ground[1];
    shadowMat[2][0] = 0.0 - light[0] * ground[2];
    shadowMat[3][0] = 0.0 - light[0] * ground[3];

    shadowMat[0][1] = 0.0 - light[1] * ground[0];
    shadowMat[1][1] = dot - light[1] * ground[1];
    shadowMat[2][1] = 0.0 - light[1] * ground[2];
    shadowMat[3][1] = 0.0 - light[1] * ground[3];

    shadowMat[0][2] = 0.0 - light[2] * ground[0];
    shadowMat[1][2] = 0.0 - light[2] * ground[1];
    shadowMat[2][2] = dot - light[2] * ground[2];
    shadowMat[3][2] = 0.0 - light[2] * ground[3];

    shadowMat[0][3] = 0.0 - light[3] * ground[0];
    shadowMat[1][3] = 0.0 - light[3] * ground[1];
    shadowMat[2][3] = 0.0 - light[3] * ground[2];
    shadowMat[3][3] = dot - light[3] * ground[3];

    glMultMatrixf((const GLfloat*)shadowMat);
}
```

Projection Shadow Trade-offs This method of shadow volume is limited in a number of ways. First, it is very difficult to use to shadow onto anything other than flat surfaces. Although you could project onto a polygonal surface, by carefully casting the shadow onto the plane of each polygon face, you would then have to clip the result to the polygon's boundaries. Sometimes depth buffering can do the clipping for you; casting a shadow to the corner of a room composed of just a few perpendicular polygons is feasible with this method.

The other problem with projection shadows is controlling the shadow's color. Since the shadow is a squashed version of the shadowing object, not the polygon being shadowed, there are limits to how well you can control the shadow's color. Since the normals have been squashed by the projection operation, trying to properly light the shadow is impossible. A shadowed polygon with an interpolated color won't shadow correctly either, since the shadow is a copy of the shadowing object.

11.4.2 Shadow Volumes

This technique treats the shadows cast by objects as polygonal volumes. The stencil buffer is used to find the intersection between the polygons in the scene and the shadow volume [20, 8, 49].

The shadow volume is constructed from rays cast from the light source, intersecting the vertices of the shadowing object, then continuing outside the scene. Defined in this way, the shadow volumes are semi-infinite pyramids, but the same results can be obtained by truncating the base of the shadow volume beyond any object that might be shadowed by it. This gives you a polygonal surface, whose interior volume contains shadowed objects or parts of shadowed objects. The polygons of the shadow volume are defined so that their front faces point out from the shadow volume itself.

The stencil buffer is used to compute which parts of the objects in the scene are in the shadow volume. It uses a non-zero winding rule technique. For every pixel in the scene, the stencil value is incremented as it crosses a shadow boundary going into the shadow volume, and decrements as it crosses a boundary going out. The stencil operations are set so this increment and decrement only happens when the depth test passes. As a result, pixels in the scene with non-zero stencil values identify the parts of an object in shadow.

Since the shadow volume shape is determined by the vertices of the shadowing object, it's possible to construct a complex shadow volume shape. Since the stencil operations will not wrap past zero, it's important to structure the algorithm so that the stencil values are never decremented past zero, or information will be lost. This problem can be avoided by rendering all the polygons that will increment the stencil count first (i.e., the front facing ones), then rendering the back facing ones.

Another issue with counting is the position of the eye with respect to the shadow volume. If the eye is inside a shadow volume, the count of objects outside the shadow volume will be -1, not zero. This problem is discussed in more detail in Section 11.4. The algorithm takes this case into account by initializing the stencil buffer to 1 if the eye is inside the shadow volume.

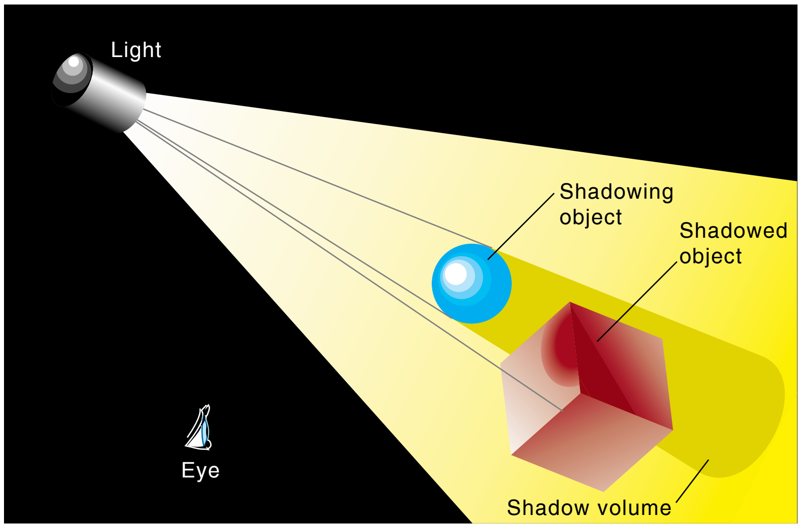


Figure 79. Shadow Volume

Here's the algorithm for a single shadow and light source:

1. The color buffer and depth buffer are enabled for writing, and depth testing is enabled.
2. Set attributes for drawing in shadow. Turn off the light source.
3. Render the entire scene.
4. Compute the polygons enclosing the shadow volume.
5. Disable the color and depth buffer for writing, but leave the depth test enabled.
6. Clear the stencil buffer to 0 if the eye is outside the shadow volume, or 1 if inside.
7. Set the stencil function to always pass.
8. Set the stencil operations to increment if the depth test passes.
9. Turn on back face culling.
10. Render the shadow volume polygons.
11. Set the stencil operations to decrement if the depth test passes.
12. Turn on front face culling.
13. Render the shadow volume polygons.
14. Set the stencil function to test for equality to 0.
15. Set the stencil operations to do nothing.
16. Turn on the light source.

17. Render the entire scene.

When the entire scene is rendered the second time, only pixels that have a stencil value equal to zero are updated. Since the stencil values were only changed when the depth test passes, this value represents how many times the pixel's projection passed into the shadow volume minus the number of times it passed out of the shadow volume before striking the closest object in the scene (after that the depth test will fail). If the shadow boundary was crossed an even number of times, the pixel projection hit an object that was outside the shadow volume. The pixels outside the shadow volume can therefore "see" the light, which is why it is turned on for the second rendering pass.

For a complicated shadowing object, it make sense to find its silhouette vertices, and use only these for calculating the shadow volume. These vertices can be found by looking for any polygon edges that either (1) surround a shadowing object composed of a single polygon, or (2) is shared by two polygons, one which is facing towards the light source, one which is facing away. You can determine which direction the polygons are facing by taking a dot product of the polygon's facet normal with the direction of the light source, or by a combination of selection and front/back face culling

Multiple Light Sources The algorithm can be extended to handle multiple light sources. For each light source, repeat the second pass of the algorithm, clearing the stencil buffer to "zero", computing the shadow volume polygons, and rendering them to update the stencil buffer. Instead of replacing the pixel values of the unshadowed scenes, choose the appropriate blending function and add that light's contribution to the scene for each light. If more color accuracy is desired, use the accumulation buffer.

The accumulation buffer can also be used with this algorithm to create soft shadows. Jitter the light source position and repeat the steps described above for multiple light sources.

Shadow Volume Trade-offs Shadow volumes can be very efficient if the shadowing object is simple. Difficulties occur when the shadowing object is a complex shape, making it difficult to compute a shadow volume. Ideally, the shadow volume should be generated from the vertices along the silhouette of the object, as seen from the light. This is not a trivial problem for complex shadowing objects.

Since the stencil count for objects in shadow depends on whether the eye point is in the shadow or not, making the algorithm independent of eye position is more difficult. One solution is to intersect the shadow volume with the view frustum, and use the result as the shadow volume. This can be a non-trivial CSG operation.

In certain pathological cases, the shape of the shadow volume may cause a stencil value underflow even if you render the front facing shadow polygons first. To avoid this problem, you can choose a "zero" value in the middle of the stencil values representable range. For an 8 bit stencil buffer, you could choose 128 as the "zero" value. The algorithm would be modified to initialize and test for this value instead of zero. The "zero" should be initialized to "zero" + 1 if the eye is inside the shadow volume.

Shadow volumes will test your polygon renderer's handling of adjacent polygons. If there are any rendering problems, such as "double hits", the stencil count can get messed up, leading to grossly incorrect shadows.

11.4.3 Shadow Maps

Shadow maps use the depth buffer and projective texture mapping to create a screen space method for shadowing objects [81, 89]. Its performance is not directly dependent on the complexity of the shadowing object.

The scene is transformed so that the eye point is at the light source. The objects in the scene are rendered, updating the depth buffer. The depth buffer is read back, then written into a texture map. This texture is mapped onto the primitives in the original scene, as viewed from the eye point, using the texture transformation matrix, and eye space texture coordinate generation. The value of the texture's texel value, the texture's "intensity", is compared against the texture coordinate's r value at each pixel. This comparison is used to determine whether the pixel is shadowed from the light source. If the r value of the texture coordinate is greater than texel value, the object was in shadow. If not, it was lit by the light in question.

This procedure works because the depth buffer records the distances from the light to every object in the scene, creating a shadow map. The smaller the value, the closer the object is to the light. The transform and texture coordinate generation is chosen so that x, y, and z locations of objects in the scene map to the s and t coordinates of the proper texels in the shadow texture map, and to r values corresponding to the distance from the light source. Note that the r values and texel values must be scaled so that comparisons between them are meaningful.

Both values measure the distance from an object to the light. The texel value is the distance between the light and the first object encountered along that texel's path. If the r distance is greater than the texel value, this means that there is an object closer to the light than this one. Otherwise, there is nothing closer to the light than this object, so it is illuminated by the light source. Think of it as a depth test done from the light's point of view.

Shadow maps can almost be done with the OpenGL 1.1 implementation. However, the ability to compare the texture's r component against the corresponding texel value is missing. There is an OpenGL extension, `SGIX_shadow`, that performs the comparison. As each texel is compared, the results set the fragment's alpha value to 0 or 1. The extension can be described as using the shadow texture r value test to mask out shadowed areas using alpha values.

Shadow Map Trade-offs Shadow maps have an advantage, being an image space technique, that they can be used to shadow any object that can be rendered. You do not have to find the silhouette edge of the shadowing object, or clip the object being shadowed. This is similar to the argument made for depth buffering vs. an object-based hidden surface removal technique, such as depth sort.

The same image space drawbacks are also true. Since the shadow map is point sampled, then mapped onto objects from an entirely different point of view, aliasing artifacts are a problem. When the texture is mapped, the shape of the original shadow texel does not necessarily map cleanly to the pixel. Two major types of artifacts result from these problems; aliased shadow edges, and self-shadowing "shadow acne" effects.

These effects ca not be fixed by simply averaging shadow map texel values. These values encode distances. They must be compared against r values, and generate a Boolean result. Averaging the texel values results in distance values that are simply incorrect. What needs to be blended are the Boolean results of the r and texel comparison. The `SGIX_shadow` extension does this, blending four adjacent comparison results to produce an alpha value. Other techniques can be used to suppress aliasing artifacts:

1. Increase shadow map/texture spatial resolution. Silicon Graphics supports off-screen buffers on some systems, called a *p-buffer*, whose resolution is not tied to the window size. It can be used to create a higher resolution shadow map.

2. Jitter the shadow texture by modifying the projection in the texture transformation matrix. The r/texel comparisons can then be averaged to smooth out shadow edges.

3. Modify the texture projection matrix so that the r values are biased by a small amount. Making the r values a little smaller is equivalent to moving the objects a little closer to the light. This prevents sampling errors from causing a curved surface to shadow itself. This r biasing can also be done with polygon offset.

One more problem with shadow maps should be noted. It is difficult to use the shadow map technique to cast shadows from a light surrounded by objects. This is because the shadow map is created by rendering the entire scene from the light's point of view. It's not always possible to come up with a transform to do this, depending on the geometric relationship between the light and the objects in the scene.

11.4.4 Soft Shadows by Jittering Lights

Most shadow techniques create a very "hard" shadow edge; surfaces in shadow, and surfaces being lit are separated by a sharp, distinct boundary, with a large change in surface brightness. This is an accurate representation for distant point light sources, but is unrealistic for many lighting environments.

A brute force way to approximate the appearance of soft shadows is to use one of the shadow techniques described above to model an area light source as a collection of point light sources. Brotman and Badler [14] used this approach with shadow volumes to generate their soft shadows.

With an accumulation buffer, you can combine the shadowed illumination from multiple point light sources. With enough point light source samples, the summed result is softer shadows, with a more gradual transition from lit to unlit areas. These soft shadows are a more realistic representation of area light sources, which create shadows consisting of an umbra (where none of the light is visible) and penumbra (where part of the light is visible).

Consider a light source that is a volume instead of merely a point. Soft shadows are created by rendering the shadowed scene multiple times, and accumulating into the accumulation buffer. Each rendering of the scene differs in that the OpenGL point position of the light source is moved slightly within the volume where the physical light being modeled would be emitting energy. To reduce aliasing artifacts, it is best to reposition the light in an irregular pattern.

Shadows from multiple, separate light sources can also be accumulated. This allows the creation of scenes containing shadows with non-trivial patterns of light and dark, resulting from the light contributions of all the lights in the scene.

11.4.5 Soft Shadows Using Textures

Heckbert and Herf describe an alternative technique for rendering soft shadows by creating a texture for each partially shadowed polygon in the scene [47]. This texture represents the effect of the scene's lights on the polygon.

For each shadowed polygon, an image is rendered which represents the contribution of each light source for each shadowed polygon, and that image is used as a texture in the final scene containing the shadowed polygon. Shadowing polygons are projected onto the shadowed polygon from the direction of the sample point on the light source. The accumulation buffer is used to average the results of that projection for several points (typically 16) on the polygon representing the light source.

The algorithm finds a single quadrilateral that tightly bounds the shadowed polygon in the plane of that polygon. The quad and the sample point on the light source are used to create a viewing frustum that projects intervening polygons onto the shadowed polygon. Multiple shadow textures per polygon are avoided because each "lighting" frustum shares the base quadrilateral, and so the shadowing results can all be accumulated into the same texture.

A pass is made for each sample point on each light source. The color buffer is cleared to the color of the light, and then the projected polygons are drawn with the ambient color of the scene. The resulting image is then added into the accumulation buffer. The final accumulation buffer result is copied into texture memory and is applied during the final scene as the polygon's texture.

Care must be taken to choose an image resolution for the shadow texture that looks acceptable on the final polygon. Depth testing and texturing can be disabled to improve performance during the projection pass. It may be necessary to save the accumulation buffer at intervals and average the results if the contribution of a shadow pass exceeds the resolution of the accumulation buffer.

A paper describing this technique in detail and other information on shadow generation algorithms is available at Heckbert and Herf's web site [48].

The Heckbert and Herf method of soft shadow generation is essentially the superposition of numerous "hard" shadows. The contributions from individual shadowing polygons are usually noticeable unless an extremely large number of shadowing polygons are used.

Soler and Sillion [91] model the special case of a light source, shadow occluder, and shadow receiver, all in parallel planes, with a convolution operation that results in shadow textures with fewer sampling-related artifacts. First, the light source and shadow occluder are represented as images. Scaled versions of the images are then efficiently convolved through the application of the Fast Fourier Transform (FFT) and its inverse to produce a soft shadow texture. Soler and Sillion go on to apply approximations that generalize the "exact" special case of parallel objects to more general situations. While the FFT at the heart of the technique is not readily accelerated by OpenGL, the technique is still very useful because of its speed for pre-computing high-quality shadow textures that can later be used as light map textures during OpenGL rendering.

12 Transparency

Transparent objects are common in everyday life and using them can add significant realism to generated scenes. In this section, we describe several techniques used to render transparent objects in OpenGL.

12.1 Screen-Door Transparency

One of the simpler transparency techniques is known as *screen-door transparency*. Screen-door transparency uses a bit mask to cause certain pixels not to be rasterized. The percentage of bits in the bitmask which are set to 1 is equivalent to the transparency of the object [27].

In OpenGL, screen-door transparency is implemented using *polygon stippling*. The command `glPolygonStipple` defines a 32x32 polygon stipple pattern. When stippling is enabled (using `glEnable(GL_POLYGON_STIPPLE)`) the low-order x and y bits of the screen coordinates of each fragment are used to index into the stipple pattern. If the corresponding bit of the stipple pattern is 0, the fragment is rejected. If the bit is 1, rasterization continues.

Since the lookup into the stipple pattern takes place in screen space, a different pattern should be used for objects which overlap, even if the transparency of the objects is the same. If the same stipple pattern is used, the same pixels in the framebuffer would be drawn for each object. Of the transparent objects, only the last (or the closest, if depth buffering is enabled) would be visible.

The biggest advantage of screen-door transparency is that the objects do not need to be sorted. Also, rasterization may be faster on some systems using the screen-door technique than using other techniques such as alpha blending. Since the screen-door technique operates on a per-fragment basis, the results will not look as smooth as if another technique had been used. However, patterns that repeat on a 2x2 grid are the smoothest and a 50% transparent "checkerboard" pattern looks quite smooth on most systems.

12.2 Alpha Blending

To draw semi-transparent geometry, the most common technique is to use *alpha blending*. In this technique, the alpha value for each fragment drawn reflects the transparency of that object. (To be totally correct, the alpha value actually represents the *opacity*, since an alpha value of 1.0 represents a 100% opaque surface). Each fragment is combined with the values in the framebuffer using the blending equation:

$$C_{out} = C_{src} * A_{src} + (1 - A_{src}) * C_{dst} \tag{11}$$

Here, C_{out} is the output color which will be written to the frame buffer. C_{src} and A_{src} are the source color and alpha, which come from the fragment. C_{dst} is the destination color, which is the color value currently in the framebuffer at the location. This equation is specified using the OpenGL command `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`. Blending is then enabled with `glEnable(GL_BLEND)`.

Transparent primitives drawn using alpha blending should *always* be drawn after all opaque primitives are drawn. Unless the transparent objects are sorted in back to front order, depth buffer updates *must* be disabled using `glDepthMask(GL_FALSE)`, although depth buffer compares should remain enabled.

If the objects are not sorted and drawn in back to front order, the above blending equation produces order-dependent rendering artifacts that can be quite objectionable. If sorting of the scene is undesirable, order dependencies can be eliminated by using `GL_ONE` for the destination factor rather than `GL_ONE_MINUS_SRC_ALPHA`. This method does not look as natural, especially when transparent objects are drawn over light objects, but it requires no sorting.

A common mistake when implementing alpha blended transparency is to assume that it requires a framebuffer with an alpha channel. The alpha value used for blended transparency comes down the graphics pipeline with each fragment; the alpha values in the framebuffer (GL_DST_ALPHA) are not actually used, so no alpha buffer is required.

The alpha value of the fragment can be set in several ways. If lighting is not being used, the alpha value can be set using a 4- component color command such as glColor4f. If lighting is enabled, the fourth color component of the diffuse reflectance coefficient of the material corresponds to the transparency of the object.

If texturing is enabled, the source of the alpha channel is controlled by the texture internal format, the texture environment function, and the texture environment constant color. The interaction is described in more detail in the glTexEnv man page. Many intricate effects can be implemented using alpha values from textures.

12.3 Sorting

The sorting step can be complicated. The sorting should be done in eye coordinates, so it is necessary to transform the geometry to eye coordinates in some fashion. If transparent objects interpenetrate, the individual triangles should be sorted and drawn from back to front. Ideally, polygons which interpenetrate should be tessellated along their intersections, sorted, and drawn independently, but this is typically not required to get good results. Frequently only crude or perhaps no sorting at all gives acceptable results.

If there is a single transparent object, or multiple transparent objects which do not overlap in screen space (i.e., each screen pixel is touched by at most one of the transparent objects), a shortcut may be taken under certain conditions. If the objects are closed, convex, and viewed from the outside, culling may be used to draw the backfacing polygons prior to the front facing polygons. The steps are as follows:

1. Enable culling: `glEnable(GL_CULL_FACE)`.
2. Configure face culling to eliminate front facing polygons: `glCullFace(FRONT)`.
3. Draw the object.
4. Configure face culling to eliminate back facing polygons: `glCullFace(BACK)`.
5. Draw the object again.
6. Disable culling: `glDisable(GL_CULL_FACE)`.

We assume that the vertices of the polygons of the object are arranged in a counter-clockwise direction when the object is viewed from the outside. If necessary, we can specify that polygons oriented clockwise should be considered front-facing with the glFrontFace command.

Drawing depth buffered opaque objects mixed with transparent objects takes somewhat more care. The usual trick is to draw the background and opaque objects first in any order with depth testing enabled, depth buffer updates enabled, and blending disabled. Next, the transparent objects are drawn from back to front with blending enabled, depth testing enabled but depth buffer updates disabled so that transparent objects do not occlude each other.

12.4 Using the Alpha Function

The *alpha function* is used to discard fragments based upon a comparison of the fragment's alpha value with a reference value. The comparison function and the reference value are specified with the command glAlphaFunc. The alpha test is enabled with `glEnable(GL_ALPHA_TEST)`.

The alpha test is frequently used to draw complicated geometry using texture maps on polygons. For example, a tree can be drawn as a picture of a tree on a single rectangle. The parts of the texture which are part of the tree have an alpha value of 1; parts of the texture which are not part of the tree have an alpha value of 0. This technique is often combined with billboarding (Section 6.10), in which a rectangle is turned to perpetually face the eye point.

Like polygon stippling, the alpha function discards fragments instead of drawing them into the framebuffer. Therefore sorting of the primitives is not necessary (unless some other mode like alpha blending is enabled). The disadvantage is that pixels must be completely opaque or completely transparent.

12.5 Using Multisampling

On systems which support the multisample extension (`SGIS_multisample`), the per-fragment sample mask may be used to change the transparency of an object. This method is basically identical to screen-door transparency described in Section 12.1, but at a sub-pixel (fragment) level.

One technique involves `GL_SAMPLE_ALPHA_TO_MASK_SGIS`. If transparent objects in a scene do not overlap, `GL_SAMPLE_ALPHA_TO_MASK_SGIS` may be used. This parameter causes the alpha of a fragment to be mapped to a sample mask which will be bitwise ANDed with the fragment's mask. The value of the generated sample mask is implementation-dependent and is a function of the pixel location and the fragment's alpha value. If two objects were drawn at the same location with the same transparency, the sample mask would be the same and the same samples would be touched. If two objects were drawn at the same location with different transparencies, results may or may not be acceptable.

The simplest technique is to use the `glSampleMaskSGIS` command to set the value of the `GL_SAMPLE_MASK_SGIS`. This value is used to generate a temporary mask which is bitwise ANDed with the fragment's mask. Again, results may not be correct if transparent objects overlap.

Currently, `SGIS_multisample` is supported by Silicon Graphics and Hewlett Packard.

13 Image Processing

13.1 Introduction

One of the strengths of OpenGL is that it provides tools for both image processing and 3D rendering. OpenGL is designed with the understanding that many image processing tools are useful for 3D graphics and vice versa. For example, convolution may be used to implement depth-of-field effects. Conversely, many operations typically thought of as image processing operations may be cast as geometric rendering and texture mapping operations. Electronic light tables (ELTs), used in defense imaging, require image transformations which can be implemented using OpenGL's textured drawing capabilities. This section demonstrates how to apply the pixel transfer pipeline, texturing, and fragment operations to the image processing problems of color manipulation, convolution, and image warping.

13.1.1 The Pixel Transfer Pipeline

The pixel transfer pipeline is the part of OpenGL most typically thought of in image processing applications. The pipeline is a configurable series of operations which are applied to each pixel during any command that moves pixels between the framebuffer, host memory, and texture memory, including:

- `glDrawPixels`
- `glReadPixels`
- `glTexImage*D`
- `glTexSubImage*D`
- `glGetTexImage*D`
- `glCopyPixels`
- `glCopyTexImage*D`
- `glCopyTexSubImage*D`

These operations move image data which falls into one of the following categories:

- Color index values
- Color values (RGBA, luminance, luminance/alpha, red, green, ...)
- Stencil buffer values
- Depth values

The "pixel transfer pipeline" processes each of these categories of data differently. For image processing, operations on color data are generally the most interesting. Before any operations are applied, source data in any color format (for example, `GL_LUMINANCE`) and type (for example, `GL_UNSIGNED_BYTE`) is converted into floating-point RGBA components. All color pixel transfer operations operate on images of this type and format. After the pixel transfer operations have been applied, the image is converted to its destination type and format.

Base OpenGL defines only a few pixel transfer operations, which are controlled using the `glPixelTransfer` command. The operations are:

- `GL_INDEX_SHIFT` and `GL_INDEX_OFFSET`, which are applied only to color index images.

- Scale and bias values which are applied to each channel of RGBA images.
- Scale and bias values which are applied to depth values.
- Pixel maps, discussed in detail in Section 13.2.3.

The pixel transfer pipeline is the part of OpenGL that has grown the most through OpenGL extensions. Some of the more interesting extensions will be discussed in this section, including the vendors who support each extension in OpenGL 1.1 as of April 1998. Where possible, we will mention techniques to achieve equivalent results on systems that do not support the extension.

13.1.2 Geometric Drawing and Texturing

OpenGL's texturing capabilities are discussed in detail in Section 6. These capabilities can be put to work to solve image processing problems. By texturing an input image onto a grid represented as geometry, we can apply arbitrary deformations to the image. Given the textured draw rates of OpenGL implementations that accelerate texturing in hardware, very impressive performance can often be achieved though the use of textured geometry. Image processing applications using texturing are discussed in Section 13.4.

13.1.3 The Framebuffer and Per-Fragment Operations

Per-fragment and framebuffer operations can be used to operate on pixels of an image in parallel. Additionally, multiple images may be combined in a variety of ways. Blending and the accumulation buffer are two areas of interest. These features are discussed in detail in Section 8. The accumulation buffer is particularly important since it provides several fundamental operations:

- Scaling of an image by a constant:
  - `glAccum(GL_MULT, <scale>)`
  - `glAccum(GL_LOAD, <scale>)`
  - `glAccum(GL_RETURN, <scale>)`
- Biasing of an image by a constant:
  - `glAccum(GL_ADD, <scale>)`
  - Clear of framebuffer with color `<scale>`, followed by `glAccum(GL_LOAD, 1)`
- Linear combination of two images on a pixel-by-pixel basis: `glAccum(GL_LOAD, <scale1>)` followed by `glAccum(GL_ACCUM, <scale2>)`

The accumulation buffer and blending are discussed in subsequent sections in terms of the image processing operations that use them.

13.1.4 The Imaging Subset in OpenGL 1.2

Several extensions to OpenGL 1.1 are incorporated as standard commands in OpenGL 1.2 as part of the optional *imaging subset*:

- Color tables (`SGI_texture_color_table` in 1.1)
- Convolution during pixel transfer (`EXT_convolution`)

- The color matrix (`SGI_color_matrix`)
- Histogram and minmax functions (`EXT_histogram`) during pixel transfer
- The blending equation and the enumerants for constant color/alpha blending, subtractive blending (`EXT_blend_subtract`), and blending with min and max operators (`EXT_blend_minmax`).

This group of extensions to the pixel transfer pipeline are useful to a class of applications that perform image processing.

The imaging subset provides color table support (`glColorTable`) in the pixel transfer pipeline before the convolution operation (`GL_COLOR_TABLE`), after convolution and before application of the color matrix (`GL_POST_CONVOLUTION_COLOR_TABLE`), and after the color matrix (`GL_POST_COLOR_TABLE`). Scale and bias are available for each color table.

The subset provides 1D, 2D and separable convolutions (`glConvolutionFilter*D` and `glSeparable-Filter2D`) in the pixel transfer pipeline, including scale and bias parameters.

Histogram and min and max functions are provided through `glHistogram` and `glMinMax`.

The imaging subset also provides support for `glBlendEquation` and `glBlendColor` and the blending modes `GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONSTANT_COLOR`, `GL_CONSTANT_ALPHA`, and `GL_ONE_MINUS_CONSTANT_ALPHA`.

If an implementation supports the imaging subset, *all* of the above features are supported. If the implementation doesn't support it, using these features will result in `GL_INVALID_OPERATION` or `GL_INVALID_ENUM`.

You can determine if an OpenGL 1.2 implementation implements the imaging subset by checking the result of `glGetString(GL_EXTENSIONS)` for the substring "`ARB_imaging`".

The imaging subset of OpenGL 1.2 is supported by the following vendors as of April, 1999:

- Silicon Graphics
- Hewlett Packard
- Sun Microsystems, Inc.
- Intergraph Computer Systems

13.1.5 Pixel Buffers

Pixel buffers (*Pbuffers* for short) are additional non-visible rendering buffers for an OpenGL renderer. The format for the color buffers and the types and sizes of any associated ancillary buffers are described similarly to the way GLX visuals are described using the GLX extension or pixel formats are described using the WGL extension. In the GLX extension, GLX visual descriptions are superseded by a newer FBconfig description which uniformly describes both visible and non-visible framebuffer resources.

Pbuffers are useful for computing and storing the results of intermediate rendering steps. The contents of a Pbuffer are transferred to a visible buffer or vice-versa using the `glxMakeCurrentRead` command to attach separate and distinct readable and writable buffers to the rendering context. Commands that read data from a buffer such as `glCopyPixels` and `glReadPixels` take source data from the currently bound read buffer and commands that generate fragments write them to the currently bound write buffer. Since Pbuffers may have ancillary buffers, it is possible for Pbuffers to store copies of the color, depth, and stencil buffers and efficiently transfer them back to the visible buffer and associated ancillary buffers.

Pixel buffers and `glxMakeCurrentRead` were originally implemented as the `SGI_make_current_read`, `SGIX_fbconfig`, and `SGIX_pbuffer` GLX extensions and were added to the GLX standard as part of the GLX 1.3 specification. Pbuffers and MakeCurrentRead functionality are available on the Windows platforms on implementations that support the `WGL_EXT_make_current_read` and `WGL_EXT_pbuffer` extensions. At the time of writing these extensions are supported by Intergraph and Silicon Graphics.

13.2 Colors and Color Spaces

This section considers ways to modify the pixels of an image on a local basis. That is, each output pixel will be a function of a single corresponding input pixel. Convolution, a non-local operation, will be considered in the next section.

13.2.1 The Accumulation Buffer: Interpolation and Extrapolation

Haeberli and Voorhies [40] have suggested several interesting image processing techniques using linear interpolation and extrapolation. Each technique is stated in terms of the formula:

$$out = (1 - x) * in_0 + x * in_1 \tag{12}$$

This equation is evaluated on a per-pixel basis. in_0 and in_1 are the input images, out is the output image, and x is the blending factor. If x is between 0 and 1, the equations describe a linear interpolation. If x is allowed to range outside $[0..1]$, the result is extrapolation [40].

In the limited case where $0 \leq x \leq 1$, these equations may be implemented using the accumulation buffer via the following steps:

1. Draw in_0 into the color buffer.
2. Load in_0, scaling by $(1 - x)$ (`glAccum(GL_LOAD, (1-x))`).
3. Draw in_1 into the color buffer.
4. Accumulate in_1, scaling by x (`glAccum(GL_ACCUM,x)`).
5. Return the results (`glAccum(GL_RETURN, 1)`).

It is assumed that in_0 and in_1 are between 0 and 1. Since the accumulation buffer can only store values in the range $[-1..1]$, for the case $x < 0$ or $x > 1$, the equation must be implemented in a different way. Given the value x, you can modify equation 12 and derive a list of accumulation buffer operations to perform the operation. Define a scale factor s such that:

$$s = max(x, 1 - x)$$

Equation 12 becomes:

$$out = s(\frac{(1-x)}{s} in_0 + \frac{x}{s} in_1)$$

and the list of steps becomes:

1. Compute s.
2. Draw in_0 into the color buffer.
3. Load in_0, scaling by $\frac{(1-x)}{s}$ (`glAccum(GL_LOAD, (1-x)/s)`).
4. Draw in_1 into the color buffer.
5. Accumulate in_1, scaling by $\frac{x}{s}$ (`glAccum(GL_ACCUM, x/s)`).
6. Return the results, scaling by s (`glAccum(GL_RETURN, s)`).

The techniques suggested by Haeberli and Voorhies use a degenerate image as in_0 and an appropriate value of x to move toward or away from that image. To increase brightness, in_0 is set to a black image and $x > 1$. To change contrast, in_0 is set to a gray image of the average luminance value of in_1. Decreasing x (toward the gray image) decreases contrast; increasing x increases contrast. Saturation may be varied using a luminance version of in_1 as in_0. (For information on converting RGB images to luminance, see Section 13.2.4.) Sharpening may be accomplished by setting in_0 to a blurred version of in_1 [40].

13.2.2 Pixel Scale and Bias Operations

Scale and bias operations can be used to adjust the colors of images. Also, they can be used to select and expand a small range of values in the input image. Scales and biases are applied at several locations in the pixel transfer pipeline. In general, scales and biases are controlled with eight floating point values (a scale and a bias for each channel).

The first scale and bias in the pixel transfer pipeline is part of base OpenGL and is specified with `glPixelTransfer(<pname>, <value>)` where `<pname>` specifies one of `GL_RED_SCALE`, `GL_RED_BIAS`, `GL_GREEN_SCALE`, `GL_GREEN_BIAS`, `GL_BLUE_SCALE`, `GL_BLUE_BIAS`, `GL_ALPHA_SCALE`, or `GL_ALPHA_BIAS`. Other sets of scale and bias values are associated with the color matrix extension (`SGI_color_matrix`) and the convolution extension (`EXT_convolution`), both of which are part of the imaging subset of OpenGL 1.2.

13.2.3 Look-Up Tables

One useful tool for color modification is the look-up table. Generally speaking, a look-up table maps an input value to a location in a table, and replaces that value with the table entry. Two look-up tables in OpenGL, pixel maps and color tables, map components independently in one-dimensional tables. These mechanisms provide efficient mapping for applications requiring no correspondence between the channels of the image. A third mechanism, pixel texturing, uses the OpenGL texturing capability to perform multi-dimensional look-ups.

Pixel Maps *Pixel maps* are a feature of base OpenGL which allow certain look-up operations to be performed. OpenGL maintains tables which map:

- The red channel to the red channel (`GL_PIXEL_MAP_R_TO_R`)
- The green channel to the green channel (`GL_PIXEL_MAP_G_TO_G`)
- The blue channel to the blue channel (`GL_PIXEL_MAP_B_TO_B`)
- The alpha channel to the alpha channel (`GL_PIXEL_MAP_A_TO_A`)
- Color indices to color indices (`GL_PIXEL_MAP_I_TO_I`)
- Stencil indices to stencil indices (`GL_PIXEL_MAP_S_TO_S`)
- Color indices to RGBA values (`GL_PIXEL_MAP_I_TO_R`, `GL_PIXEL_MAP_I_TO_G`, `GL_PIXEL_MAP_I_TO_B`, and `GL_PIXEL_MAP_I_TO_A`)

Tables that map color indices to RGBA values are used automatically whenever an image with a color index format is transferred to a destination which requires an RGBA image. For example, performing a `glDrawPixels` of a color index image to an RGBA framebuffer would result in application of the I to RGBA pixel maps. Other tables are enabled with the commands `glPixelTransfer(GL_MAP_COLOR, 1)` and `glPixelTransfer(GL_MAP_STENCIL, 1)`.

Pixel maps are defined using the `glPixelMap` command and queried using the `glGetPixelMap` command. Details on the use of these commands may be found in [12]. The sizes of the pixel maps are not tied together in any way. For example, the R to R pixel map does not need to be the same size as the G to G pixel map.

Each system provides a constant, `GL_MAX_PIXEL_MAP_TABLE`, which gives the maximum size of a pixel map which may be defined.

The Color Table Extension The color table extension, `SGI_color_table`, provides additional look-up tables in the OpenGL pixel transfer pipeline. Although the capabilities of color tables and pixel maps are similar, the semantics are different.

The color table extension defines the following look-up tables:

- "First" color table (`GL_COLOR_TABLE_SGI`)
- Post convolution color table (`GL_POST_CONVOLUTION_COLOR_TABLE_SGI`)
- Post color matrix color table (`GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI`)

Each table is independently enabled and disabled using the `glEnable` and `glDisable` commands. One, two, or all three of the tables may be applied during the same operation. Color index images have to be converted to RGBA images using the I to RGBA pixel maps described in the previous section before they can be passed through the RGBA portion of the pixel transfer pipeline.

Color tables are specified using the `glColorTableEXT` and `glCopyColorTableEXT` commands and are queried using the `glGetColorTableEXT` command. The man pages for these commands provide details on their use. Note that unlike the RGBA to RGBA pixel maps, all channels of a color table are specified at the same time.

When a color table is specified, an internal format parameter (for example, `GL_RGB` or `GL_LUMINANCE_EXT`) gives the channels present in the table. When the color table is applied to an image (which is by definition RGBA), channels of the image which are not present in the color table are left unmodified. In this way, color tables are more flexible than pixel maps, which replace all channels of the input image.

Although color tables provide similar functionality to pixel maps and may prove more useful in certain circumstances, they do not replace pixel maps in the OpenGL pipeline and the tables managed by pixel maps and color tables are independent. It is possible to apply both a pixel map and a color table (or color tables) during the same pixel operation (although the utility of this is questionable). The maximum sizes and relative efficiencies of pixel maps and color tables vary from platform to platform.

The color table extension in OpenGL 1.1 is supported by the following vendors:

- Silicon Graphics
- Hewlett Packard
- Sun Microsystems, Inc.

The Texture Color Table Extension The texture color table extension (`SGI_texture_color_table`) provides a color table (`GL_TEXTURE_COLOR_TABLE_SGI`) which is applied to texels after filtering and prior to combination with the fragment color with the texture environment operation. The procedures to define, enable, and disable the texture color table are the same as those of the tables in `SGI_color_table`.

The texture color table extension is currently supported by the following vendors:

- Silicon Graphics
- Evans & Sutherland
- Hewlett Packard
- Sun Microsystems, Inc.

The texture color table is *not* part of the imaging subset of OpenGL 1.2.

The Pixel Texture Extension The pixel texture extension (`SGIX_pixel_texture`) allows multi-dimensional lookups through OpenGL's texturing capability. Remember that OpenGL defines rasterization of a pixel image during a `glDrawPixels` or `glCopyPixels` command as the generation of a fragment for each pixel in the image. Per-fragment operations are applied, including texturing (if enabled). If the input image contained color data, each fragment's color comes from the color of the pixel that generated it. The texture coordinate of the fragment is taken from the current raster position, which is generally not useful because the texture coordinate will be constant over the pixel rectangle. The pixel texture extension allows the texture coordinates s, t, q, and r of the fragment to be copied from the color coordinates R, G, B, and A of the pixel. With three and four dimensional textures (`EXT_texture3D` and `SGIS_texture4D`), arbitrary effects can be implemented (although the texture storage requirements to do so can be staggering).

The pixel texture extension is supported by the following vendors:

- Silicon Graphics

Pixel texture is *not* part of the imaging subset of OpenGL 1.2.

Equivalent Functionality Without `SGIX_pixel_texture` There is no way to apply a true multidimensional lookup to a pixel image without `SGIX_pixel_texture`. In some cases, pixel maps and color tables may be used as a substitute. Blending, accumulation buffer operations, or scale/bias operations may be used when the function to be applied is linear and each channel is independent. In other cases, the application will have to perform the lookup on the host or draw a textured point for each pixel in the image.

13.2.4 The Color Matrix Extension

The color matrix extension (`SGI_color_matrix`) defines a 4x4 color matrix which is managed using the same commands as the projection, modelview, or texture matrix. The color matrix premultiplies RGBA colors in the pixel transfer pipeline and as such can be used to perform linear color space conversions.

Since the color matrix is treated like any other matrix, it is always enabled and defaults to the identity matrix. To change the contents of the color matrix, the current matrix mode must be set to `GL_COLOR` using `glMatrixMode`. After that, the color matrix may be manipulated using the same commands as any other matrix; for example, `glLoadMatrix`, `glPushMatrix`, and `glPopMatrix`.

The color matrix extension is currently supported on the following platforms:

- Silicon Graphics

Equivalent Functionality Without `SGI_color_matrix` Unfortunately, the functionality of `SGI_color_matrix` is difficult to efficiently duplicate on systems which do not support the extension. In the case where the image is going from the host to the framebuffer (a `glDrawPixels` operation), the best way to handle the situation is the split the image up into red, green, blue, and alpha images (via application processing or a draw followed by reads with `format` set to `GL_RED`, `GL_GREEN`, `GL_BLUE`, or `GL_ALPHA`). The red, green, blue, and alpha images can be drawn as `GL_LUMINANCE` images. RGBA scale operations are applied, with the four values equal to the row of the matrix corresponding to source channel. The images are composited in the framebuffer using blending (`glBlendFunc(GL_ONE, GL_ONE)`).

Scale and Bias Scale and bias operations may be performed using the color matrix. A scale factor can be applied using the `glScale` command. A bias is equivalent to a translation and may be applied using the `glTranslate` command. Using `glScale` and `glTranslate`, the R scale or bias is put in the x parameter, the G scale or bias in the y parameter, and the B scale or bias in the z parameter. Modifications to the A channel must be specified using `glLoadMatrix` or `glMultMatrix`. In general, using the color matrix to implement scale and bias will be slower than using a transfer operation which implements scale and bias directly, but management of state may be easier using color matrices. Also, the scale and bias could be rolled into another color matrix operation.

Conversion to Luminance Converting a color image into a luminance image may be accomplished by scaling each component by its weight in the luminance equation.

$$\begin{bmatrix} L \\ L \\ L \\ 0 \end{bmatrix} = \begin{bmatrix} R_w & G_w & B_w & 0 \\ R_w & G_w & B_w & 0 \\ R_w & G_w & B_w & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \\ A \end{bmatrix}$$

The recommended weight values for R_w, G_w, and B_w are 0.3086, 0.6094, and 0.0820. Some authors have used the values from the YIQ color conversion equation (0.299, 0.587, and 0.114), but Haeberli notes that these values are incorrect in a linear RGB color space.[39]

Modifying Saturation The *saturation* of a color is the distance of that color from a gray of equal intensity.[27] Haeberli has suggested modifying saturation using the equation:

$$\begin{bmatrix} R' \\ G' \\ B' \\ A \end{bmatrix} = \begin{bmatrix} a & d & g & 0 \\ b & e & h & 0 \\ c & f & i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \\ A \end{bmatrix}$$

where:

$$\begin{aligned} a &= (1-s) * R_w + s \\ b &= (1-s) * R_w \\ c &= (1-s) * R_w \\ d &= (1-s) * G_w \\ e &= (1-s) * G_w + s \\ f &= (1-s) * G_w \\ g &= (1-s) * B_w \\ h &= (1-s) * B_w \\ i &= (1-s) * B_w + s \end{aligned}$$

with R_w, G_w, and B_w as described in the above section. Since the saturation of a color is the difference between the color and a gray value of equal intensity, it is comforting to note that setting s to 0 gives the luminance equation. Setting s to 1 leaves the saturation unchanged; setting it to -1 takes the complement of the colors [39].

Hue Rotation Changing the hue of a color may be accomplished by loading a rotation about the gray vector $(1, 1, 1)$. This operation may be performed in one step using the `glRotate` command. The matrix may also be constructed via the following steps [39]:

1. Load the identity matrix (`glLoadIdentity`).
2. Rotate such that the gray vector maps onto the z axis using the `glRotate` command.
3. Rotate about the z axis to adjust the hue (`glRotate(<degrees>, 0, 0, 1)`).
4. Rotate the gray vector back into position.

Unfortunately, a naive application of `glRotate` will not preserve the luminance of the image. To avoid this problem, you must make sure that areas of constant luminance map to planes perpendicular to the z axis when you perform the hue rotation. Recalling that the luminance of a vector (R, G, B) is equal to:

$$(R, G, B) \cdot (R_w, G_w, B_w)$$

you realize the plane of constant luminance k is defined by:

$$(R, G, B) \cdot (R_w, G_w, B_w) = k$$

Therefore, the vector (R_w, G_w, B_w) is perpendicular to planes of constant luminance. The algorithm for matrix construction becomes the following [39]:

1. Load the identity matrix.
2. Apply a rotation matrix M such that the gray vector $(1, 1, 1)$ maps onto the positive z axis.
3. Compute $(R'_w, G'_w, B'_w) = M(R_w, G_w, B_w)$. Apply a skew transform which maps (R'_w, G'_w, B'_w) to $(0, 0, B'_w)$. This matrix is:
$$\begin{bmatrix} 1 & 0 & \frac{-R'_w}{B'_w} & 0 \\ 0 & 1 & \frac{-G'_w}{B'_w} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
4. Rotate about the z axis to adjust the hue.
5. Apply the inverse of the shear matrix.
6. Apply the inverse of the rotation matrix.

It is possible to compute a single matrix as a function of R_w, G_w, B_w, and the degrees of rotation which performs this operation.

CMY Conversion The CMY color space describes colors in terms of the subtractive primaries: cyan, magenta, and yellow. CMY is used mainly for hardcopy devices such as color printers. Generally, the conversion from RGB to CMY follows the equation [27]:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

CMY conversion may be performed using the color matrix or a scale and bias operation. The conversion is equivalent to a scale by -1 and a bias by $+1$. Using the 4x4 color matrix, the equation may be restated as:

$$\begin{bmatrix} C \\ M \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \\ 1 \end{bmatrix}$$

Here, the incoming alpha channel must be equal to 1. If the source is RGB, the 1 will be added automatically in the format conversion stage of the pipeline.

A related color space, CMYK, uses a fourth channel (K) to represent black. Since conversion to CMYK requires a $min()$ operation, it cannot be performed using the color matrix.

The extension `EXT_CMYKA` also supports conversion to and from CMYK and CMYKA. This extension is currently supported by Evans & Sutherland.

YIQ Conversion The YIQ color space is used in U.S. color television broadcasting. Conversion from RGBA to YIQA may be accomplished using the color matrix:

$$\begin{bmatrix} Y \\ I \\ Q \\ A \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 & 0 \\ 0.596 & -0.275 & -0.321 & 0 \\ 0.212 & -0.523 & 0.311 & 0 \\ 0.000 & 0.000 & 0.000 & 1 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \\ A \end{bmatrix}$$

(Generally, YIQ is not used with an alpha channel so the fourth component is eliminated.) The inverse matrix is used to map YIQ to RGBA [27].

13.3 Convolutions

13.3.1 Introduction

Convolutions are used to perform many common image processing operations including sharpening, blurring, noise reduction, embossing, and edge enhancement. This section begins with a very brief overview of the mathematics of the convolution operation. More detailed explanations of the mathematics and uses of the convolution operation can be found in many books on computer graphics and image processing such as [27]. After this brief mathematical introduction, this section will describe two ways to perform convolutions using OpenGL: via the accumulation buffer and via the convolution extension.

13.3.2 The Convolution Operation

The *convolution operation* is a mathematical operation which takes two functions $f(x)$ and $g(x)$ and produces a third function $h(x)$. Mathematically, convolution is defined as:

$$h(x) = f(x) * g(x) = \int_{-\infty}^{+\infty} f(\tau)g(x-\tau)d\tau \tag{13}$$

$g(x)$ is referred to as the *filter*. The integral only needs to be evaluated over the range where $g(x-\tau)$ is nonzero (called the *support* of the filter).[27]

In spatial domain image processing, you discretize the operation. $f(x)$ becomes an array of pixels $F[x]$. The kernel $g(x)$ is an array of values $G[0...(width-1)]$ (assume finite support). Equation 13 becomes:

$$H[x] = \sum_{i=0}^{width-1} F[x+i]G[i] \tag{14}$$

Two-Dimensional Convolutions Since you generally operate on two-dimensional images in image processing, extend Equation14 to:

$$H[x][y] = \sum_{j=0}^{height-1} \sum_{i=0}^{width-1} F[x+i][y+j]G[i][j] \tag{15}$$

During convolution, the value for a pixel in the output image is calculated by aligning the filter array (kernel) with the pixel at the same location in the input image and summing the values of the pixels in the input array multiplied by the corresponding values in the filter array.

The algorithm can be visualized as a loop over the width and height of the input image. In the loop, the filter is typically centered over each input pixel. Another loop over the width and height of the filter multiplies the values in the filter array with the values under the filter in the input image. The results of the multiplication are added together and stored in the output image in the same $< x, y >$ location as the pixel in the input image. The output and input images are kept logically separate so that the results of one step in the loop don't affect later steps in the loop.

The convolution filter may have a single element per-pixel, where the RGBA components are scaled by the same value, or the filter may have separate red, green, blue, and alpha values for each element.

Separable Filters In the general case, the two-dimensional convolution operation requires $(width * height)$ multiplications for each output pixel. Separable filters are a special case of general convolution in which the filter

$$G[0..(width - 1)][0..(height - 1)]$$

can be expressed in terms of two vectors

$$G_{row}[0..(width - 1)]G_{col}[0..(height - 1)]$$

such that for each $(i, j)\epsilon([0..(width - 1)], [0..(height - 1)])$

$$G[i][j] = G_{row}[i] * G_{col}[j]$$

If the filter is separable, the convolution operation may be performed using only $(width + height)$ multiplications for each output pixel. Applying the separable filter to Equation15 becomes:

$$H[x][y] = \sum_{j=0}^{height-1} \sum_{i=0}^{width-1} F[x+i][y+j]G_{row}[i]G_{col}[j]$$

Which can be simplified to:

$$H[x][y] = \sum_{j=0}^{height-1} G_{col}[j] \sum_{i=0}^{width-1} F[x+i][y+j]G_{row}[i]$$

To apply the separable convolution, first apply G_{row} as though it were a $width$ by 1 filter. Then apply G_{col} as though it were a 1 by $height$ filter.

13.3.3 Convolutions Using the Accumulation Buffer

The convolution operation may be implemented by building the output image in the accumulation buffer. For each kernel entry $G[i][j]$, translate the input image by $(-i, -j)$ from its original position and then accumulate the translated image using the command `glAccum(GL_ACCUM, G[i][j])`. This translation can be performed by `glCopyPixels` but an application may be able to more efficiently redraw the image shifted using `glViewport`. $width * height$ translations and accumulations must be performed. Skip clearing the accumulation buffer by using `GL_LOAD` instead of `GL_ACCUM` for the first accumulation.

Here is an example of using the accumulation buffer to convolve using a Sobel filter, commonly used to do edge detection. This filter is used to find horizontal edges:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Since the accumulation buffer can only store values in the range (-1..1), first modify the kernel such that at any point in the computation the values do not exceed this range:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = 4 * \begin{bmatrix} \frac{-1}{4} & \frac{-2}{4} & \frac{-1}{4} \\ 0 & 0 & 0 \\ \frac{1}{4} & \frac{2}{4} & \frac{1}{4} \end{bmatrix}$$

The operations needed to apply the filter are:

1. Draw the input image.
2. `glAccum(GL_LOAD, 1/4)`
3. Translate the input image left by one pixel.
4. `glAccum(GL_ACCUM, 2/4)`
5. Translate the input image left by one pixel.
6. `glAccum(GL_ACCUM, 1/4)`
7. Translate the input image right by two pixels and down by two pixels.
8. `glAccum(GL_ACCUM, -1/4)`
9. Translate the input image left by one pixel.
10. `glAccum(GL_ACCUM, -2/4)`
11. Translate the input image left by one pixel.
12. `glAccum(GL_ACCUM, -1/4)`
13. Return the results to the framebuffer (`glAccum(GL_RETURN, 4)`).

In this example, each pixel in the output image is the combination of pixels in the 3 by 3 pixel square whose lower left corner is at the output pixel. At each step, the image is shifted so that the pixel that would have been under the kernel element with the value used is under the lower left corner. As an optimization, ignore locations where the kernel is equal to zero.

A general algorithm for the 2D convolution operation is:

```
Draw the input image
for (j = 0; j < height; j++) {
  for (i = 0; i < width; i++) {
    glAccum(GL_ACCUM, G[i][j]*scale);
    Move or redraw the input image to the left by 1 pixel
  }
  Move or redraw the input image to the right by width pixels
  Move or redraw the input image down by 1 pixel
}
glAccum(GL_RETURN, 1/scale);
```

`scale` is a value chosen to ensure that the intermediate results cannot go outside a certain range. In the Sobel filter example, `scale = 4`. Assuming the input values are in $(0..1)$, `scale` can be naively computed using the following algorithm:

```
float minPossible = 0, maxPossible = 1;
for (j = 0; j < height; j++) {
  for (i = 0; i < width; i++) {
    if (G[i][j] < 0) {
       minPossible += G[i][j];
    } else {
       maxPossible += G[i][j];
    }
  }
}
scale = 1.0 / ((-minPossible > maxPossible) ?
               -minPossible : maxPossible);
```

Since the accumulation buffer has limited precision, more accurate results can be obtained by changing the order of the computation and computing `scale` accordingly. Additionally, if values in the input image can be constrained to a smaller range, `scale` can be made larger, which may also give more accurate results.

For separable kernels, convolution can be implemented using $width + height$ image translations and accumulations. A general algorithm is:

```
Draw the input image
for (i = 0; i < width; i++) {
  glAccum(GL_ACCUM, Grow[i] * rowScale);
  Move or redraw the input image to the left 1 pixel
}
glAccum(GL_RETURN, 1 / rowScale);
for (j = 0; j < height; j++) {
  glAccum(GL_ACCUM, Gcol[j] * colScale);
  Move or redraw the framebuffer image down by 1 pixel
}
glAccum(GL_RETURN, 1 / colScale);
```

In this example, it is assumed that scales for the row and column filters have been determined in a similar fashion to the general two-dimensional filter, such that the accumulation buffer values will never go out of range.

13.3.4 The Convolution Extension

The *convolution extension*, `EXT_convolution`, defines a stage in the OpenGL pixel transfer pipeline which applies a 1D, separable 2D, or general 2D convolution. The 1D convolution is applied only to 1D texture downloads and is infrequently used. 2D kernels are specified using the commands `glConvolutionFilter2DEXT`, `glCopyConvolutionFilter2DEXT`, and `glSeparableFilter2DEXT`. The convolution stage is enabled using the enumerant `GL_CONVOLUTION_2D_EXT` or `GL_SEPARABLE_2D_EXT`. Filters are queried using `glGetConvolutionFilterEXT` and `glGetSeparableFilterEXT`.

The maximum permitted convolution size is machine-dependent and may be queried using `glGetConvolutionParameterfvEXT` with the parameters `GL_MAX_CONVOLUTION_WIDTH_EXT` and `GL_MAX_CONVOLUTION_HEIGHT_EXT`.

The relative performance of separable and general filters varies from platform to platform, but it is best to specify a separable filter whenever possible.

`EXT_convolution` is currently supported by the following vendors:

- Silicon Graphics
- Hewlett Packard
- Sun Microsystems, Inc.

13.3.5 Useful Convolution Filters

This section briefly describes several useful convolution filters. The filters may be applied to an image using either the convolution extension or the accumulation buffer technique. Unless otherwise noted, the kernels presented are normalized (that is, the kernel weights sum to 0).

You should keep in mind that this section is intended only as a very basic reference. Numerous texts on image processing provide more details and other filters including [66].

Line detection Detection of one pixel wide lines can accomplished with the following filters:

Horizontal Edges

$$\begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix}$$

Vertical Edges

$$\begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix}$$

Left Diagonal Edges

$$\begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}$$

Right Diagonal Edges

$$\begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix}$$

Gradient Detection (Embossing) Changes in value over 3 pixels can be detected using kernels called *Gradient Masks* or *Prewitt Masks*. The direction of the change from darker to lighter is described by one of the points of the compass. The 3x3 kernels are as follows:

North

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

West

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

East

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

South

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Northeast

$$\begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix}$$

Smoothing and Blurring Smoothing and blurring operations are low-pass spatial filters. They reduce or eliminate high-frequency aspects of an image.

Arithmetic Mean The arithmetic mean simply takes an average of the pixels in the kernel. Each element in the filter is equal to 1 divided by the total number of elements in the filter. Thus the 3x3 arithmetic mean filter is:

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

Basic Smooth: 3x3 (not normalized)

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Basic Smooth: 5x5 (not normalized)

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 4 & 4 & 4 & 1 \\ 1 & 4 & 12 & 4 & 1 \\ 1 & 4 & 4 & 4 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

High-pass Filters A high-pass filter enhances the high-frequency parts of an image. This type of filter is used to sharpen images.

Basic High-Pass Filter: 3x3

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Basic High-Pass Filter: 5x5

$$\begin{bmatrix} 0 & -1 & -1 & -1 & 0 \\ -1 & 2 & -4 & 2 & -1 \\ -1 & -4 & 13 & -4 & -1 \\ -1 & 2 & -4 & 2 & -1 \\ 0 & -1 & -1 & -1 & 0 \end{bmatrix}$$

Laplacian Filter The *Laplacian* is used to enhance discontinuities. The 3x3 kernel is:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

and the 5x5 is:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 24 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Sobel Filter The *Sobel filter* consists of two kernels which detect horizontal and vertical changes in an image. If both are applied to an image, the results can by used to compute the magnitude and direction of the edges in the image. If the application of the Sobel kernels results in two images which are stored in the arrays `Gh[0..(height-1][0..(width-1)]` and `Gv[0..(height-1)][0..(width-1)]`, the magnitude of the edge passing through the pixel `x, y` is given by:

$$M_{sobel}[x][y] = \sqrt{Gh[x][y]^2 + Gv[x][y]^2} = |Gh[x][y]| + |Gv[x][y]|$$

(you are justified in using the magnitude representation since the values represent the magnitude of orthogonal vectors). The direction can also be derived from `Gh` and `Gv`:

$$\phi_{sobel}[x][y] = tan^{-1}(\frac{Gv[x][y]}{Gh[x][y]})$$

The 3x3 Sobel kernels are:

Horizontal

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Vertical

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

13.3.6 Correlation and Feature Detection

The *correlation* operation is defined mathematically as:

$$h(x) = f(x) \circ g(x) = \int_{-\infty}^{+\infty} f^*(\tau)g(x+\tau)d\tau \qquad (16)$$

The $f^*(\tau)$ is the complex conjugate of $f(\tau)$, but since this section will discuss correlation for signals which only contain real values, substitute $f(\tau)$.

Correlation is useful for feature detection; applying correlation to an image that possibly contains a target feature and an image of that feature forms local maxima or pixel value “spikes” in candidate positions. This is useful in detecting letters on a page, or the position of armaments on a battlefield. Correlation can also be used to detect motion, such as the velocity of hurricanes in a satellite image or the jittering of an unsteady camera.

For two-dimensional discrete images, you may use Equation 15 to evaluate correlation.

The convolution extension (`EXT_convolution`) in OpenGL may be used to apply correlation to an image, but only for features no larger than the maximum convolution kernel size. For larger images or platforms which do not supply the convolution extension, use the accumulation buffer technique for convolution. (It is worth the effort to consider an alternative method, such as applying a multiplication in the frequence domain [35], if your feature and candidate images are very large.)

Once you have applied convolution, your application will need to find the "spikes" to determine where features have been detected. To aid this process, it may be useful to apply thresholding with a color table (`SGI_color_table`) to convert candidates pixels to one value and non-candidates to another.

One method used for finding features uses the following steps:

- Draw a small image containing just the feature to detect.
- Create a convolution filter containing that image.
- Transfer the image to the convolution filter using `glCopyConvolutionFilter2DEXT`.
- Draw your candidate image into the color buffers.
- Optionally configure a threshold for candidate pixels:
  - Create a color table using `glColorTableSGI`.
  - `glEnable(GL_POST_CONVOLUTION_COLOR_TABLE_SGI)`.
- `glEnable(GL_CONVOLUTION_2D_EXT)`
- Apply pixel transfer to your candidate image using `glCopyPixels`.
- Read back the frame buffer using `glReadPixels`.
- Measure candidate pixel locations.

If your candidate image comes from a source other than the OpenGL color buffer, use `glDrawPixels` to apply the pixel transfer pipeline to your image.

If features in the candidate image are not pixel-exact, for example if they are rotated slightly or blurred, it may be necessary to create the feature image using jittering and blending, and then lower the acceptance threshold in the color table.

13.4 Image Warping

13.4.1 The Pixel Zoom Operation

OpenGL provides control over the generation of fragments from pixels via the pixel zoom operation. Zoom factors are specified using `glPixelZoom`. Negative zooms are used to specify reflections.

Pixel zooming may prove faster than the texture mapping techniques described below on some systems, but do not provide as fine a control over filtering.

13.4.2 Warps Using Texture Mapping

Image warping or dewarping may be implemented using texture mapping by defining a correspondence between a uniform polygonal mesh and a warped mesh. The points of the warped mesh are assigned the corresponding texture coordinates of the uniform mesh and the mesh is rendered texture mapped with the original image. Using this technique, simple transformations such as zoom, rotation, or shearing can be efficiently implemented. The technique also easily extends to much higher-order warps such as those needed to correct distortion in satellite imagery.

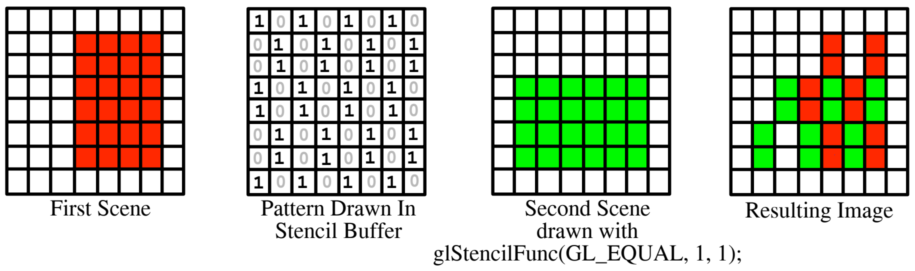

First Scene
Pattern Drawn In Stencil Buffer
Second Scene drawn with glStencilFunc(GL_EQUAL, 1, 1);
Resulting Image

Figure 80. Using Stencil to Dissolve Between Images

14 Special Effects

14.1 Dissolves with Stencil

Stencil buffers can be used to mask selected pixels on the screen. This allows for pixel by pixel compositing of images. You can draw geometry or arrays of stencil values to control, per pixel, what is drawn into the color buffer. One way to use this capability is to composite multiple images.

A common film technique is the "dissolve", where one image or animated sequence is replaced with another, in a smooth sequence. The stencil buffer can be used to implement arbitrary dissolve patterns. The alpha planes of the color buffer and the alpha function can also be used to implement this kind of dissolve, but using the stencil buffer frees up the alpha planes for motion blur, transparency, smoothing, and other effects.

The basic approach to a stencil buffer dissolve is to render two different images, using the stencil buffer to control where each image can draw to the framebuffer. This can be done very simply by defining a stencil test and associating a different reference value with each image. The stencil buffer is initialized to a value such that the stencil test will pass with one of the images' reference values, and fail with the other. An example of a dissolve part way between two images is shown in Figure 80.

At the start of the dissolve (the first frame of the sequence), the stencil buffer is all cleared to one value, allowing only one of the images to be drawn to the framebuffer. Frame by frame, the stencil buffer is progressively changed (in an application defined pattern) to a different value, one that passes only when compared against the second image's reference value. As a result, more and more of the first image is replaced by the second.

Over a series of frames, the first image "dissolves" into the second, under control of the evolving pattern in the stencil buffer.

Here is a step-by-step description of a dissolve.

1. Clear the stencil buffer with `glClear(GL_STENCIL_BUFFER_BIT)`.
2. Disable writing to the color buffer, using `glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE)`.
3. If the values in the depth buffer should not change, use `glDepthMask(GL_FALSE)`.

For this example, we'll have the stencil test always fail, and set the stencil operation to write the reference value to the stencil buffer. Your application will also need to turn on stenciling before you begin drawing the dissolve pattern.

1. Turn on stenciling; `glEnable(GL_STENCIL_TEST)`.

2. Set stencil function to always fail; `glStencilFunc(GL_NEVER, 1, 1)`.
3. Set stencil op to write 1 on stencil test failure; `glStencilOp(GL_REPLACE, GL_KEEP, GL_KEEP)`.
4. Write the dissolve pattern to the stencil buffer by drawing geometry or using `glDrawPixels`.
5. Disable writing to the stencil buffer with `glStencilMask(GL_FALSE)`.
6. Set stencil function to pass on 0; `glStencilFunc(GL_EQUAL, 0, 1)`.
7. Enable color buffer for writing with `glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE)`.
8. If you're depth testing, turn depth buffer writes back on with `glDepthMask`.
9. Draw the first image. It will only be written where the stencil buffer values are 0.
10. Change the stencil test so only values that are 1 pass; `glStencilFunc(GL_EQUAL, 1, 1)`.
11. Draw the second image. Only pixels with stencil value of 1 will change.
12. Repeat the process, updating the stencil buffer, so that more and more stencil values are 1, using your dissolve pattern, and redrawing image 1 and 2, until the entire stencil buffer has 1's in it, and only image 2 is visible.

If each new frame's dissolve pattern is a superset of the previous frame's pattern, image 1 doesn't have to be re-rendered. This is because once a pixel of image 1 is replaced with image 2, image 1 will never be redrawn there. Designing the dissolve pattern with this restriction can improve the performance of this technique.

14.2 Motion Blur

This is probably one of the easiest effects to implement. Simply re-render a scene multiple times, incrementing the position and/or orientation of an object in the scene. The object will appear blurred, suggesting motion. This effect can be incorporated in the frames of an animation sequence to improve its realism, especially when simulating high-speed motion.

The apparent speed of the object can be increased by dimming its blurred path. This can be done by accumulating the scene without the moving object, setting the value parameter to be larger than $1/n$. Then re-render the scene with the moving object, setting the value parameter to something smaller than $1/n$. For example, to make a blurred object appear 1/2 as bright, accumulated over 10 scenes, do the following:

1. Render the scene without the moving object, using `glAccum(GL_LOAD,.5f)`.
2. Accumulate the scene 10 more times, with the moving object, using `glAccum(GL_ACCUM,.05f)`.

Choose the values to ensure that the non-moving parts of the scene retain the same overall brightness.

It's also possible to use different values for each accumulation step. This technique could be used to make an object appear to be accelerating or decelerating. As before, ensure that the overall scene brightness remains constant.

If you are using motion blur as part of a real-time animated sequence, and your value is constant, you can improve the latency of each frame after the first n dramatically. Instead of accumulating n scenes, then discarding the image and starting again, you can subtract out the first scene of the sequence, add in the new one, and display the result. In effect, you're keeping a "running total" of the accumulated images.

The first image of the sequence can be "subtracted out" by rendering that image, then accumulating it with `glAccum(GL_ACCUM, -1.f/n)`. As a result, each frame only incurs the latency of drawing two scenes; adding in the newest one, and subtracting out the oldest.

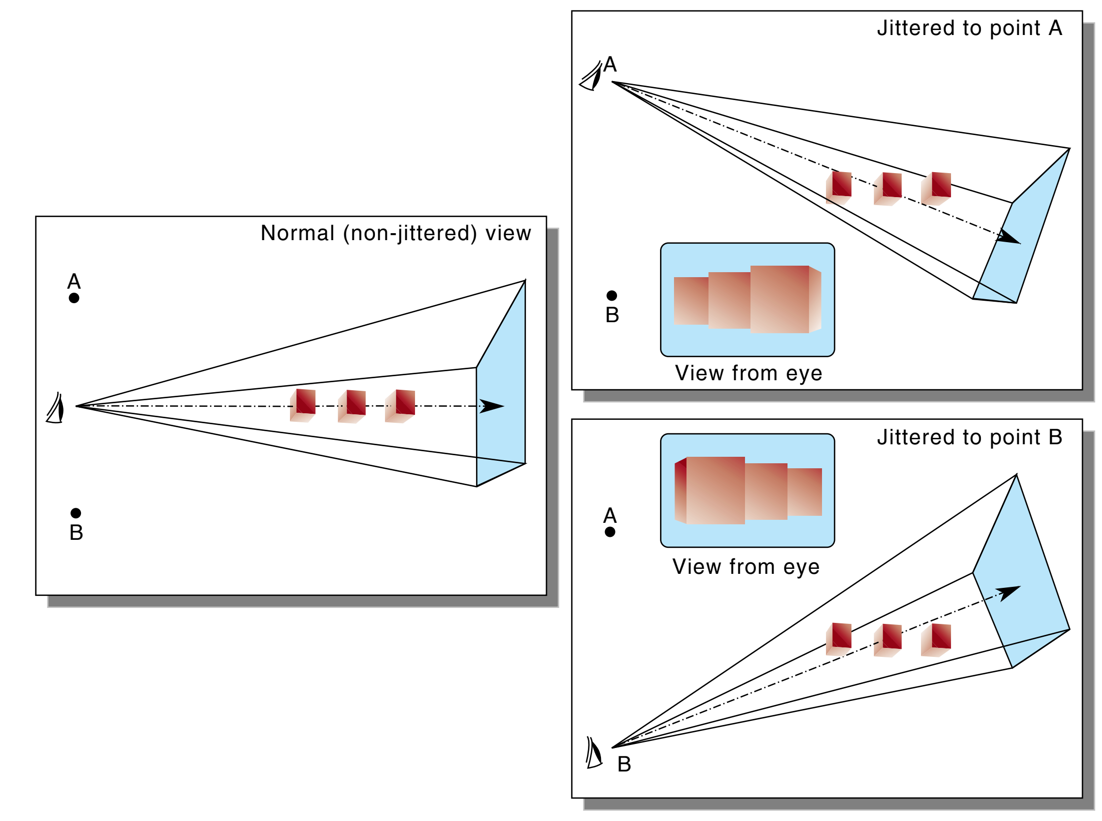


Figure 81. Jittered Eye Points

14.3 Depth of Field

OpenGL's perspective projections simulate a pinhole camera; everything in the scene is in perfect focus. Real lenses have a finite area, which causes only objects within a limited range of distances to be in focus. Objects closer or farther from the camera are progressively more blurred.

The accumulation buffer can be used to create depth of field effects by jittering the eye point and the direction of view. These two parameters change in concert, so that one plane in the frustum doesn't change. This distance from the eye point is thus in focus, while distances nearer and farther become more and more blurred.

To create depth of field blurring, the perspective transform changes described for antialiasing in Section 9.5 are expanded somewhat. This code modifies the frustum as before, but adds in an additional offset. This offset is also used to change the modelview matrix; the two acting together change the eye point and the direction of view:

```
void frustum_depthoffield(GLdouble left, GLdouble right,
                          GLdouble bottom, GLdouble top,
                          GLdouble near, GLdouble far,
                          GLdouble xoff, GLdouble yoff,
                          GLdouble focus)
{
    glFrustum(left - xoff * near/focus,
                 right - xoff * near/focus,
                top - yoff * near/focus,
                bottom - yoff * near/focus,
                near, far);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(-xoff, -yoff);
}
```

The variables `xoff` and `yoff` now jitter the eye point, not the entire scene. The focus variable describes the distance from the eye where objects will be in perfect focus. Think of the eye point jittering as sampling the surface of a lens. The larger the lens, the greater the range of jitter values, and the more pronounced the blurring. The more samples taken, the more accurate a sampling of the lens. You can use the jitter values given in Section 9.5.

This function assumes that the current matrix is the projection matrix. It sets the frustum, then sets the modelview matrix to the identity, and loads it with a translation. The usual modelview transformations could then be applied to the modified modelview matrix stack. The translate would become the last logical transform to be applied.

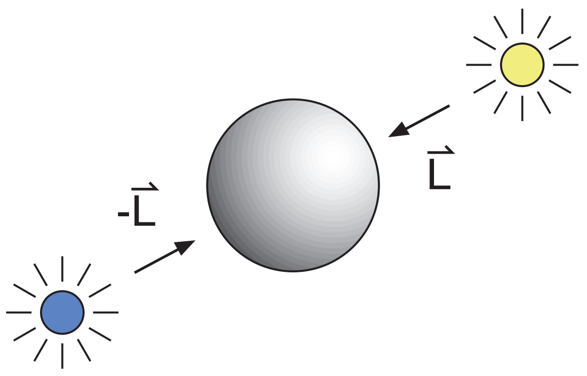


Figure 82. Opposing Lights Approximating Warm to Cool Shift

15 Illustration and Artistic Techniques

In applications such as scientific visualization and technical illustration photorealism detracts from rather than enhances the information in the rendered image. Applications such as cartography and computer aided design benefit from the use of hidden surface elimination and 3D illumination and shading techniques, but the goal of increased insight from the generated images suggests some different processing compared to those used to achieve photorealism.

15.1 Non-photorealistic Lighting Models

In [36, 37] lighting and shading algorithms are developed based on traditional technical illustration practices. A non-photorealistic lighting model for matte and metal surfaces is constructed. The model for matte surfaces use both luminance and hue changes to indicate surface orientation. This lighting model reduces the dynamic range of the luminance, reserving luminance extremes to emphasize edges and highlights. To compensate for the reduced dynamic range and provide additional shape cues, tone-based shading adds hue shifts to the lighting model. Exploiting the relationship that cool colors (blue, violet, green) recede and warm colors (red, orange, yellow) advance, a sense of depth is added by including cool to warm color transitions in the model. The diffuse cosine term is replace with the term:

$$d_m((\frac{1+\vec{N}\cdot\vec{L}}{2})d_{l_{cool}} + (1 - \frac{1+\vec{N}\cdot\vec{L}}{2})d_{l_{warm}})$$

where $d_{l_{cool}}$, and $d_{l_{warm}}$ are linear combinations of a cool color (e.g., a shade of blue) combined with the object's diffuse reflectance and a warm color (e.g., yellow) also combined with the object's diffuse reflectance. A typical value for $d_{l_{cool}}$ is $(0., 0., .4) + .2d_m$ and for $d_{l_{warm}}$ is $(.4, .4, 0.) + .6d_m$

This modified diffuse lighting model can be approximated using OpenGL lighting by using two opposed lights ($\vec{L}$, $-\vec{L}$) as shown in Figure 82. The diffuse colors are set to $(d_{l_{warm}} - d_{l_{cool}})/2$, and $(d_{l_{cool}} - d_{l_{warm}})/2$, respectively, and the ambient color set to $(d_{l_{cool}} + d_{l_{warm}})/2$ and the specular and emissive contributions set to zero. Objects are drawn with the material reflectances set to one (white). Highlights can be added in a subsequent pass using blending to accumulate the result. Alternatively the environment mapping techniques discussed in Section 10.4 can be used to capture and apply the BRDF at the expensive of computing a map for each different object material.

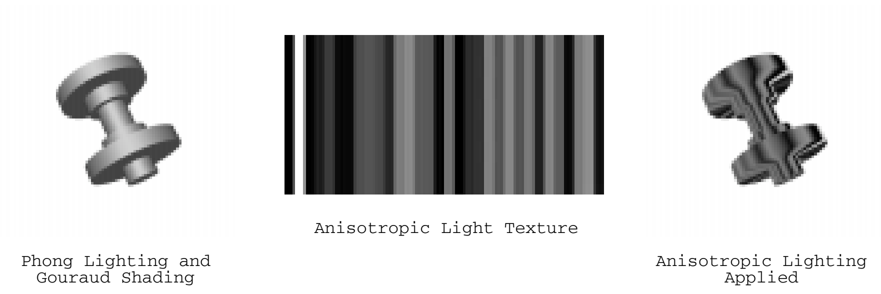

Anisotropic Light Texture
Phong Lighting and Gouraud Shading
Anisotropic Lighting Applied

Figure 83. Simulation of Anisotropic Lighting

For metallic surfaces, the lighting model is further augmented to simulate the appearance of anisotropic reflection (Section 10.9). While anisotropic reflection typically occurs on machined (milled) metal parts rather than polished parts, the anisotropic model is used to provide a cue that the surfaces are metal and to provide a sense of curvature. To simulate the anisotropic reflection pattern, the curved surface is shaded with stripes along the parametric axis of maximum curvature. The intensity of the stripes are random values between 0.0 and 0.5, except the stripe closest to the light source is set to 1.0 to simulate a highlight. The values between the stripes are interpolated. This process is implemented in the OpenGL pipeline using texture mapping. A small one- or two-dimensional luminance texture is created containing the randomized set of stripe values. The stripe at s coordinate zero (or some well known position) is set to the value one. The object is drawn with texture enabled and the wrap mode set to `GL_CLAMP` and the s texture coordinate set to vary along the curvature. The position of the highlight is adjusted by biasing the s coordinate with the texture matrix. This procedure is illustrated in Figure 83.

15.2 Edge Lines

An important aspect of the lighting model is reducing the dynamic range of the luminance to make edges and highlights more distinct. Edges can be further emphasized by outlining the silhouette and boundary edges in a dark color. Some algorithms for drawing silhouette lines are described in Section 7.4. Additional algorithms using image processing techniques are described in [85] and can be implemented using the OpenGL pipeline as described in Section 13. [37, 62] discuss software methods for extracting silhouette edges which can then be drawn as lines.

15.3 Gradual Cutaway Views

Engineering drawings of complex objects (such as automobiles) may show a cutaway view, removing some layers of the object (such as the outer shell) in order to reveal the object's inner components and their respective positions. When the purpose of the drawing is more sales-oriented, the cutaway view may be done in a more artistic style, with the cut edge of object's outer shell faded gradually and the parts of the edge closer to the viewer becoming more and more transparent. Additional stylistic touches can be added by showing the seams of the object shell, and have them also fade to transparency at a slightly different rate than the shell surface itself.

This effect can be done in a straightforward way using OpenGL. This technique uses texture mapping and texture coordinate generation to modulate the alpha component of an object's shell. The object must be divided into two parts that can be rendered separately; the object's shell and the object's interior. The interior is rendered first in a standard fashion, using depth buffering. The object shell is rendered, but a one-dimensional texture map containing an alpha component ramp is used to modulate the object color.

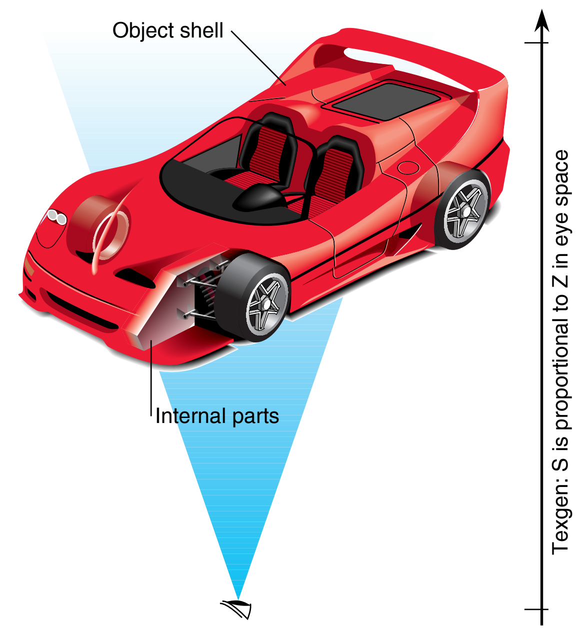


Figure 84. Gradual Cutaway Using a 1D Texture

If alpha blending is enabled, using `glBlendMode(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`, the texture map will scale down the alpha component of the shell as it gets closer to the viewer, rendering it more transparent. The edges of the shell can be rendered as a separate pass, using a slightly different 1D texture map or different texgen plane equation to produce a different rate of transparency change from that of the shell surface.

Since the shell itself is blended, it must be handled as a transparent object to avoid render order artifacts. Both depth buffering and alpha blending using `GL_SRC_ALPHA` for the source and `GL_ONE_MINUS_SRC_ALPHA` for the destination require depth sorted primitives in order to work reliably. The shell should be sorted so the surfaces more distant from the viewer are rendered first. If the shell is convex, and the surface primitives are oriented consistently, an easy way to do this is with face culling. If the shell primitives are oriented to be outward facing, rendering the shell twice, first with front face, then back face culling will draw the surfaces in the proper depth order. For more information, see Section 12 in these course notes.

15.3.1 Steps to Generating a Cutaway Shell

1. Draw the object internals with depth buffering.
2. Enable and configure a 1 dimensional texture ramp; use `GL_ALPHA` as the format.
3. Enable and configure texture coordinate generation for the s component; use eye linear, and set the s eye plane to map $-z$ over the range of the object shell cutaway from 0 to 1.
4. Enable blending, and set the blend mode: source is `GL_SRC_ALPHA`, destination is `GL_ONE_MINUS_SRC_ALPHA`.
5. Render the shell of the object in depth order; most distant objects first. For convex shells, this could be done using face culling.

6. Load a different texture ramp in the 1D texture map.
7. Render the shell edges; you can do this by re-rendering the shell after call `glPolygonMode` with the mode set to `GL_LINE`.

If you want to render the shell edges, you'll need to use polygon offset, or some other method, such as using the stencil buffer, to avoid z fighting. A reasonable setting to try would be `glPolygonOffset(-1.f, -1.f)`.

15.3.2 Refinements

There are a number of parameters you will want to adjust for maximum effect. One is the shape of the texture ramp for both the shell and the shell edges. A linear ramp produces a somewhat abrupt cutoff; tapering the beginning and end of the ramp will produce a smoother transition. The texture ramps can also be adjusted by changing the texgen s eye plane. Changing the plane values can move the distance and the range of the cutaway transition zone.

Since both the shell and the interior of the object will be lit, there is some question as to what the back surface of the shell revealed by the cutaway should look like. As before, aesthetics and the surrounding scene will determine what's best. Some choices would be showing the back of the shell in a darker version of the shell's color, unlit. Another possibility is to use back face lighting on the shell's interior.

15.3.3 Rendering a Surface Textured Shell

The steps above assume an untextured object shell. If the shell itself has a surface texture, things get more involved. The preference would be to apply both the 2D surface texture and the 1D transparency texture ramp simultaneously. In order to blend two textures together, use a multipass method. The basic idea is to separate the blend function `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)` into two separate steps. There are now three objects to consider; internal components of the object, the shell of the object textured with a surface texture, and the shell of the object textured with the 1D alpha texture. The alpha textured shell is used to adjust the transparency of the other two objects separately.

Two approaches suggest themselves, based on your hardware's capabilities. If your system supports an alpha buffer, the approach is only a little more complicated. If you don't, you can do it with two buffers.

Alpha Buffer Approach You render the internal object as before, then adjust the transparency of the resulting image by rendering the alpha-textured shell with the blend mode set to `glBlendFunc(GL_ZERO, GL_ONE_MINUS_SRC_ALPHA)`. The alpha values from the shell are used to scale the image of the object internals that have been rendered into the framebuffer. The alpha values themselves are also saved into the alpha buffer.

Now depth buffer update is disabled, and the surface textured shell is rendered, with the blend mode set to `glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE)`. In this way the internal part of the object, which has already been scaled by $1 - srcalpha$ is summed with the surface textured shell, which is blended by $1-(1-srcalpha) = srcalpha$, giving the desired result.

1. Configure a window that can store alpha color values.
2. Draw the object internals with depth buffering.
3. Mask off depth buffer updates.
4. Enable blend mode.
5. `glBlendFunc(GL_ZERO, GL_ONE_MINUS_SRC_ALPHA)`
6. Draw alpha textured shell to adjust internal objects' transparency.

7. `glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE)`
8. Disable 1D Texturing Enable 2D texturing.
9. Render surface textured shell.

No Alpha Buffer Approach If you don't have an alpha buffer to store intermediate alpha values, then you'll have to render two images, one of the internal objects, one of the surface textured shell, then combine the two images using blending.

The first steps are the same as the alpha buffer approach: You render the internal object as before, then adjust the transparency of the resulting image by rendering the alpha textured shell with the blend mode set to `glBlendFunc(GL_ZERO, GL_ONE_MINUS_SRC_ALPHA)`. The alpha values from the shell are used to scale the image of the object internals that have been rendered into the framebuffer. This time the alpha values are lost.

In a separate buffer (or different area of the window) Render the surface textured shell. Now adjust the transparency of this image by re-rendering the shell using only the alpha texture. This time the blend mode should be `glBlendFunc(GL_ZERO, GL_SRC_ALPHA)`. This image now has it's transparency adjusted.

Now you can combine the two images using `glCopyPixels` with the blend function set to `glBlendFunc(GL_ONE, GL_ONE)`. This brings the two halves of the blend operation together.

There is one problem. There is no depth testing between the transparent shell and the internal objects images. You can also take care of this using a stenciling technique. The technique allows you, in effect, copy an image with its depth information. The stencil buffer is used to save the results of depth comparing the two images' depth values, and used as a per-pixel mask to control the merging of the two images. See Section 8.7 for details.

15.4 Depth Cuing

Perspective projection and hidden surface and line elimination are regularly used to add a sense of depth to rendered images. However, other kinds of depth cues are useful, particularly for applications using orthographic projections. The term depth-cuing is typically associated with the technique of changing the intensity of an object as a function of distance from the eye. This effect is typically implemented using the fog stage of the OpenGL pipeline. For example, using a linear fog function with the fog color set to black results in a linear interpolation between the object's color and zero, where the interpolation factor, f, is determined by the distance of each fragment from the eye, $f = \frac{end-z}{end-start}$

It is also straightforward to implement a cuing algorithm using a 1D texture map using `glTexGen` to generate a texture coordinate using a linear texture coordinate generation function to compute a coordinate proportional to the distance from the eye along the z-axis. The filtered texel value is used as the interpolation factor between the polygon color and texture environment color. One advantage of using a 1D texture is that the map can be used to encode an arbitrary function of distance which can be used to implement more extreme cuing effects. Textures can also be useful on OpenGL implementations that use per-vertex rather than per-pixel fog calculations.

Other types of depth cues may also be useful. Section 17.7 describes methods for generating points with appropriate perspective foreshortening. Similar problems exist for line primitives as their width is specified in window coordinates rather than object coordinates. For most wireframe display applications this is not an issue since the lines are typically very narrow. However, for some applications wider lines are used to convey other types of information. A simple method for generating perspective lines is to use polygonal primitives rather than lines.

15.5 Cross Hatching and 3D Halftones

In [85], Saito suggests using cross-hatching to shade 3D geometry to provide visual cues. Rather than performing 2D hatching using a fixed screen space pattern (e.g., using polygon stipple) an algorithm is suggested for generat-

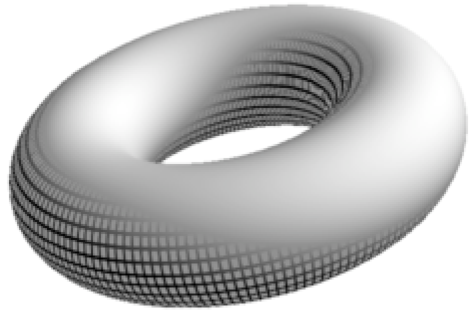


Figure 85. 3D Cross Hatching

ing hatch lines aligned with parametric axes for the object (for example a sequence of straight lines traversing the length of a cylinder, or a sequence of rings around a cylinder).

A similar type of shading can be achieved using texture mapping. The parametric coordinates of the object are used as the texture coordinates at each vertex and a 1D or 2D texture map consisting of a single stripe is used to generate the hatching. This method is similar to the methods for generating contour lines in Section 6.14, except the iso-contours are now lines of constant parametric coordinate. If a 1D texture is used, at minimum two alternating texels are needed. A wrap mode of `GL_REPEAT` is used to replicate the stripe pattern across the object. If a 2D texture is used then the texture map contains a single stripe. Two parametric coordinates can be cross hatched at the same time using a 2D texture map with stripes in both the s and t directions. To reduce artifacts, the object needs to be tessellated finely enough to provide accurate sampling of the parametric coordinates.

This style of shading can be useful with bilevel output devices. For example, a luminance hatched image can be thresholded against an unlit version of the same image using a max function. This results in the darker portions of the shaded image being hatched, while the brighter portions remain unchanged as shown in Figure 85. This idea is generalized to the notion of a 3D halftone screen in [41].

Traditionally, halftones are generated by thresholding an image against a halftone screen. Graphics devices such as laser printers can approximate the variable width circles used in halftones by using circular raster patterns. Such patterns can be generated using a clustered-dot ordered dither [27]. An $n \times n$ dither pattern can be represented as a matrix. For dithering operations in which the number of output pixels is greater than the number of input pixels i.e., each input pixel is converted to a $n \times n$ set of output pixels, the input pixel is compared against each element in the dither matrix. For each element that the input pixel is larger than the dither element, a 1 is output, otherwise a 0. An example 3×3 dither matrix is:

$$D = \begin{pmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{pmatrix}$$

A dithering operation of this type can be implemented using the OpenGL pipeline as follows

1. Replicate the dither patter in the framebuffer to generate an a threshold image the size of the output image. Use `glCopyPixels` to perform the replication.
2. Set `glPixelZoom(n,n)` to replicate each pixel to a $n \times n$ block.
3. Move the threshold image into the accumulation buffer with `glAccum(GL_LOAD,1.0)`.
4. Use `glDrawPixels` to transfer the expanded source image in the framebuffer

5. Call `glAccum(GL_ACCUM,-1.0)`.
6. Call `glAccum(GL_RETURN,-1.0)` to invert and return the result.
7. Set up `glPixelMap` to map 0 to 0 and everything else to 1.0.
8. Call `glReadPixels` with the pixel map to retrieve the thresholded image.

Alternatively, the subtractive blend function can be used to do the thresholding instead of the accumulation buffer if the imaging extensions are present. If the input image is not a luminance image, it can be converted to luminance using the techniques described in Section 13.2.4 during the transfer to the framebuffer. If the framebuffer is not large enough to hold the output image, the source image can be split into tiles which are processed separately and merged.

15.6 2D Drawing Techniques

While most applications use OpenGL for rendering 3D data it is inevitable that 3D geometry must be combined with some 2D screen space geometry. OpenGL is designed to coexist with other renderers in the window system, that is, OpenGL and other renderers can operate on the same window. For example, X Window System 2D drawing primitives and OpenGL commands can be combined together in a window. Similarly, Win32 GDI drawing and OpenGL commands can be combined together in the same window.

One advantage of using the native window system 2D renderer is that the 2D renderers typically provide more control over 2D operations. For example, control over the joins in lines (mitre, round, bevel), the end caps on lines (round, butt) and have rasterization rules that are somewhat easier to predict. For example, both the X Window System and Win32 GDI have very precise specifications of the algorithms for rasterizing 2D lines, whereas OpenGL has provided some latitude for implementors which occasionally causes problems with application portability.

Some disadvantages in not using OpenGL commands for 2D rendering, are that the native window system 2D renderers are not tightly integrated with the OpenGL renderers. For example, the 2D renderer typically does not update or test against the depth or other ancillary buffers, the coordinate system typically has the origin at the top left corner of the window, some desirable OpenGL functionality may not be available in the 2D renderer (e.g., framebuffer blending, antialiased lines), and the 2D code is less portable.

To specify object coordinates in screen space, an orthographic projection is used. For a window of width w and height h, the transformation maps object coordinate $(0, 0)$ to window coordinate $(0, 0)$ and object coordinate (w, h) to window coordinate (w, h). Since OpenGL has pixel centers on half integer boundaries, this mapping results in pixel centers at 0.5, 1.5, 2.5, ..., w-.5 along the x-axis and 0.5, 1.5, 2.5, ..., h-.5 along the y-axis.

One difficulty is that the line rasterization rules for OpenGL are designed to avoid multiple pixel hits when drawing connected line primitives. The reason for this is that multiple hits cause difficulties in using blending or stenciling algorithms to merge multiple primitives reliably. This means that if a rectangle is drawn with a `GL_LINE_LOOP`, the rectangle will be properly closed with no missing pixels, whereas if the same rectangle is drawn with a `GL_LINE_STRIP`, or independent `GL_LINES` there will likely be pixels missing and or multiple hits on the rectangle boundary at or near the vertices of the rectangle. A second issue is that OpenGL does use half integer pixel centers, whereas the native window system specifies pixel centers at integer boundaries. Application developers often incorrectly use integer pixel centers with OpenGL without compensating in the projection transform.

15.6.1 Line Joins

Wide lines in OpenGL are drawn by expanding the width of the line along the x or y direction of the line for y-major and x-major lines, respectively (a line is x-major if the slope is in the range $[-1, 1]$). When two non-colinear wide lines are connected together the overlap in the end caps leaves a noticeable gap. In 2D drawing engines such

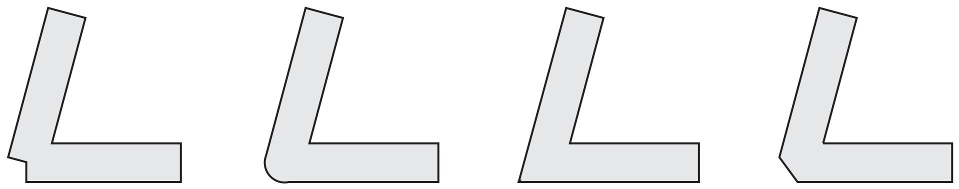


Figure 86. Line Join Styles: None, Round, Miter, Bevel

as GDI or the X Window System, lines can be joined using a number of different styles: round, mitered, or beveled as shown in Figure 86.

A round join can be implemented by drawing a round antialiased point with a size equal to the line width at the shared vertex. For most implementations the antialiasing algorithm generates a point that is similar enough in size to match the line width without noticeable artifacts. However, many implementations do not support large antialiased point sizes, making it necessary to use a triangle fan or texture mapped quadrilateral to implement a disc of the desired radius to join very wide lines.

A mitered join can be implemented by drawing a triangle fan with the first vertex at the shared vertex of the join, and three remaining vertices at the two outside vertices of rectangles enclosing the two lines, and the intersection point of the two outside edges of the wide lines extended until they meet. For an x-major line of width w and window coordinate end points (x_0, y_0) and (x_1, y_1) the rectangle around the line is $(x_0, y_0 - (w-1)/2)$, $(x_0, y_0 - (w-1)/2 + w)$, $(x_1, y_1 - (w-1)/2 + w)$, $(x_1, y_1 - (w-1)/2)$.

Mitered joins with very sharp angles are not aesthetically pleasing, so for angles less then some threshold angle (typically 11 degrees) a bevel join is used. A bevel join can be constructed by rendering a single triangle consisting of the shared vertex and the two outside corner vertices of the lines as described above.

Having gone this far, it is a small step to switch from using lines to using triangle strips to draw the lines instead. One advantage of using lines is that many OpenGL implementations support antialiasing up to moderate line widths, whereas there is substantially less support for polygon antialiasing. Wide antialiased lines can be combined with antialiased points to do round joins, but requires the overlap algorithm from Section 7.5 to sort the coverage values. Accumulation buffer antialiasing can be used with triangle primitives as well.

15.7 Painting on Images

In [42], Haeberli describes a technique for using filters in the form of brush strokes to create abstract images (impressionistic paintings) from source images. The output image is generated by rendering an ordered list of brush strokes. Each brush stroke contains color, shape, size, and orientation information. Typically the color information is determined by sampling the corresponding location in the source image. The size, shape, and orientation information are generated from user input in an interactive painting application. The paper also describes some novel algorithms for generating brush stroke geometry. One example is the use of a depth buffered cone with the base of the cone parallel to the image plane at each stroke location. This algorithm results in a series of Dirichlet domains [80] where the color of each domain is sampled from the source image.

Additional effects can be achieved by preprocessing the input image. For example, the contrast can be enhanced, or the image sharpened using simple image processing techniques. Edge detection operators can be used to recover paths for brush strokes to follow. These operations can be automated and combined with stochastic methods to choose brush shape and size to generate brush strokes automatically.

16 Scientific Visualization Techniques

16.1 Scalar Field Visualization

Scalar field visualization is the graphical expression of relationships between scalar values distributed in space.

The difficulty of rendering scalar fields depends on a number of factors; The dimensionality of the data, whether the data has been sampled on a regular grid or not, and the range of values the field can assume.

16.1.1 Definition of a Scalar Field

A scalar value is a single component that can assume one of a range of values. An example of this is temperature. In contrast, a vector value has more than one component. An example of a vector value is a direction, composed of x, y, and z components. A scalar field is an arrangement of scalar values distributed in a space.

Typically, scalar field data being visualized is not continuous (as the original scalar field is), but is composed of a set of discrete sampled values. The sample spacing may be regular, forming a grid, or the sample spacing might be irregular, with varied spacing between samples values.

The sample values themselves will be limited to some finite range, due to limitations in the measuring equipment or restrictions on the simulation that created the values. Both the range of possible sample values, and the significance of the values themselves, vary with the application. For example, if a scalar field of atmospheric temperature values is used for aviation flight planning, the range of values is bounded to the values possible in the atmosphere. If the field is used for flight planning in winter, the values of greatest interest are right around the freezing point of water, since airframe icing is a major concern. In summer, higher temperatures mean lower aircraft performance, so unusually high temperatures are of greater interest.

When visualizing data, the exact values of the data are not as important as the relationship between values. Data visualization is used to gain insight into the data set, and expose relationships between values that might not be apparent in the raw data. As a result, intuitive, but less exact, representations of data values are often used.

16.1.2 Representing Data Values

There are a number of ways to represent scalar data values for visualization. The most obvious is color. A set of color values can be used to represent the range of values the data can assume. There are a few issues that must be considered when choosing color values. First, there are large set of color values that can be used to represent the color range. The colors should be chosen to make the values intuitive. A common technique is to use “cold” colors such as black, purple, blue for low values, and “hot” colors such as red, orange, and white to represent high values. Sometimes green hues are used to fill in the mid-range values. This ties into viewer expectations about heat and color, indicating the “energy” of the data values.

Colors tailored to the application space will give the most mileage. If the goal is to notice data discrepancies, make those values stand out from the rest of the color range. For example, unusual values can be shaded with red hues, with the rest of the value range shades of green. The color range can be made consistent by changing only the hue, fixing the saturation and brightness to only slow changes over the entire data range.

In general, chose one or more “color paths” through RGB color space to represent the data range, taking psychological and application specific factors into account.

Once colors are chosen, there are a number of ways to render them. The most obvious way is to map the data values to RGB values directly in the application. OpenGL can be used to simplify and accelerate the process and reduce the amount of work performed in the application.

One obvious way is to use color index values, choosing a colormap where data values index the desired colors. This approach is not recommended, however, for a number of reasons. First, it is a limited approach. Color index values must be integers, and the allowable range of values is limited by the maximum size of the colormap.

Second, graphics API implementations are moving away from color index support, and many implementors don't emphasize the color index part of their implementations. Color index applications may be unaccelerated, and possibly not as well implemented as the RGBA path.

Fortunately, there is a better way to provide a mapping between data values and rgb colors: texture maps. The texture map can provide a mapping between data values, input as texture coordinates, to colors, which are mapped as colors in the texture map itself. Texture mapping is optimized and hardware accelerated on almost every implementation of OpenGL at the time of writing, and arbitrarily large mappings can be created using one or more texture maps.

A simple example might clarify this technique. Imagine a set of 50 data values to map colors onto:

1. Create a 1D 64 entry texture map (you're limited to creating textures whose dimensions are powers-of-2)
2. Load the first 50 entries of the texture with the color values corresponding to the first 50 data values.
3. Choose a texture transformation matrix that maps the data values to the S values that index the corresponding texels. This will work if the data values can be mapped with a linear or perspective transformation to the appropriate texture coordinates. If your implementation supports a lookup table associated with texture filtering, such as `GL_TEXTURE_COLOR_TABLE_SGI`, you can use the `glColorTableSGI` command to create an arbitrary non-linear mapping between data value and texture coordinate. If you don't have this support, you can create a lookup table in the application.
4. If you don't want to interpolate between data values, use `GL_NEAREST` for `GL_TEXTURE_MIN_FILTER`, and `GL_TEXTURE_MAG_FILTER`, and use the same texture coordinates over all the vertices of the primitive you want to color. Remember that OpenGL indexs a texel by truncating the texture coordinate scaled by the texture size. This is written into the specification, so you can always look up the exact texel you want.
5. If you do want to interpolate colors between data values, use `GL_LINEAR` for `GL_TEXTURE_MIN_FILTER` and `GL_TEXTURE_MAG_FILTER`, and choose the correct data values at the sample points, letting texture filtering do the color interpolation for you. Since OpenGL texture mapping is perspective correct, you don't have to worry about perspective projection coloring artifacts.

16.1.3 Interpolating Data Values

Be careful about interpolating between sampled scalar values. Most interpolation techniques assume a linear relationship between sampled data values, which might not be the case. A linearly interpolated set of colors applied to non-linear data can create misleading and incorrect images. There are two basic approaches to avoiding this problem. First, don't interpolate between data values at all. Only display the data values at the sample points. This results in no interpolation problems, but possibly makes the data harder to interpret. The other possibility is to ensure adequate sampling. If the data is known to be linear between sample points, within an acceptable error, then interpolation can be safely done. What "acceptable error" means depends on the application.

16.1.4 Rendering Data Values

Keep in mind that the purpose of scientific visualization is to visualize relationships in the data, not show every value with precision. It isn't necessary to know the exact value of a data point by looking at an image of it. It is important to be able to see how that data point relates to others. Annotation can quickly reach the point of clutter and confusion. If the application user requests additional information about the data, the application can provide interactive manipulation and picking techniques to alter the mapping and make queries about the data in the image.

A few rendering techniques follow. Many more are available.

Point Fields Point fields display data positions as a "cloud" of points in space. The data values themselves can be displayed as point color, point size, or point shape. Points can be rendered very quickly with OpenGL. If the data is static, display lists will give the best performance for most applications. If the data is dynamically changing, try using vertex arrays to represent the data. This also can simplify the application, since data value changes can be accessed through arrays. Some data can be represented by changing the point size, but this can be expensive on some implementations. It may also help performance to sort points by size before rendering. This technique is also used to render natural phenomena in Section 17.7. If the number of points is small enough, and the implementation has high enough performance, individual billboards can be used instead of OpenGL points. This technique, as described in Section 6.10, allows points of different shapes, and high quality appearance can be achieved though alpha blending the polygon edges.

Height Fields If the spatial distribution of the points is two dimensional, the third dimension can be use to represent the data value of each sample. One way to create the geometry for this technique is to perturb a flat plane, changing the altitude of the plane at each sample point. This perturbation can be used to distort the plane into a mountainous surface, using the data points as vertices of the polygon. Another technique is to leave the plane flat, and draw lines perpendicular from the surface to the data point. In each case, the data value is the perpendicular distance from the point to the reference plane. If the data is very noisy, the distorted plane can become very jagged, making it hard to interpret the data. Additional points can be inserted to smooth transitions, and the plane can be made partially transparent so that the terrain won't obscure neighboring points as much. For details on rendering transparent objects, see Section 12.2

This technique can be more render intensive than point fields. Back face culling may improve fill performance if the viewing angle is oblique enough that many triangles are backfacing. As before, a display list for a static surface, or vertex array representation for a dynamic one, will also improve performance.

How the perturbed plane is tessellated can effect both rendering performance and visual appearance of the resulting surface. If the samples are regularly spaced, choosing connectivity is relatively easy. If the the sample spacing isn't regular, then a deluanay tessellation scheme is a good choice, since it produces "fat" triangles (triangles with large angles at each vertex), which gives the best representation for a given surface. Delaunay triangulation algorithms are beyond the scope of these notes; an excellent book on the subject is written by O'Rourke [73] this and similar topics. See Section 3 for a discussion of tessellation.

Contouring Contouring is closely related to height fields. If the data values are organized into regions, the relationship can be represented as a 3D contour map. Each data value divides up the terrain into regions, bounded by isolines. Each region is displaced a different height above the reference surface. The boundaries of each data value can be stitched together and tessellated, producing a 3D surface. Alternatively, the region can be sampled, and rendered as a height field. In both this and the previous technique, the elevation data can be delineated with color values on the surface, as with topological maps. This can be easily rendered with a one-dimensional texture and the use of the texture generation functionality in OpenGL:

1. Create a 1D texture map with the desired color values.
2. Configure texgen to map different elevation values to texture coordinates (e.g. map y values to s coordinate values).
3. Render the surface with 1D texturing and texgen enabled.

You probably want to use `GL_EYE_LINEAR` texture generation, so the surface can be manipulated relative to the viewer without changing the texture mapping.

Annotating Metrics A variation on the contouring idea, in [95], Teschner proposes a method for displaying metrics, such as 2D tick marks, on an object using a 2D texture map containing the metrics. Texture coordinates are generated as a distance from object coordinates to a reference plane. For the 2D case, two reference planes are used. An example application for this technique is to create a 2D texture marked off with tick marks every kilometer in both the s and t directions and map this texture on to terrain data using a `GL_REPEAT` texture coordinate wrap mode. An `GL_OBJECT_LINEAR` texture coordinate generation mode is used, with the reference planes at $x = 0$ and $z = 0$ and a scale factor set such that a vertex coordinate which is 1km from the $x - y$ or $z - y$ plane produces a texture coordinate value equal to the distance between two tick marks in texture coordinate space.

16.1.5 Regular vs. Irregular Data Sampling

Regular data sampling makes the creation of geometry much easier for surface representations, however many real-world data sets are irregularly sampled. As mentioned previously, an irregular sampling array becomes a tessellation problem.

There may be a strong temptation to resample the data into a regular grid. This approach can be problematic in many cases, since the data may not change linearly between grid points. False relationships can be created in the resampled data.

Regridding can also lead to a performance degradation, since you will probably end up with a larger number of data points.

16.1.6 3D Scalar Fields

Visualizing a three dimensional field of data points is more challenging than 2D for two reasons: First, producing surfaces to represent data values is no longer easy, since there isn't a free dimension to use. Second, data values tend to obscure each other, making it harder to see relationships in the data.

Point fields, as described above, can be used to visualize 3D data fields. If the point density is high, points can be rendered with transparency, and sorted from back to front. This makes the obscured points more visible. See Section 12.2 for details on transparency.

Another technique is to use volume visualization techniques to view the data. This technique is especially useful for very dense datasets. See Section 16.2 for details.

Another method is to calculate isosurfaces from the dataset. Data points with similar values can be connected together with a tessellated surface. These surfaces, like point fields, should be drawn partially transparent, using alpha blending, and rendered from back to front. Each isosurface can have a separate color, to make them easier to distinguish. See O'Rourke's book [73] for ideas on generating the iso surfaces.

16.1.7 Multiple Scalar Fields

Rather than visualize a single data set, you may want to go a step farther and explore the interrelationships betweens two more more scalar fields. This can be done, at the expense of visual complexity, by tagging data values with multiple independent attributes. Some possibilities are:

1. point field: color for one data value, transparency for the other
2. point field: color for one data value, point size for the other
3. point field: color for one data value, texture pattern for the other
4. point field: two complementary color scales that can be mixed to result in unique color values
5. height field: height for one data value, color for the other

Of course, sharing attributes on sample points only works if both data sets were sampled at the same locations. If not, there will be two interacting sets of independent sample points, which will be hard to decipher visually, especially if the data set is dense.

16.1.8 Manipulating Scalar Fields

Interactivity is an important feature available in all of these data visualization techniques. A visualization system becomes dramatically more useful if the data set can be manipulated by the viewer, especially in real-time.

Some important interaction features include:

1. 3D manipulation of the data space *The viewer should be able to rotate, scale, and zoom the data space. This makes it possible to clarify the relationship between data points, especially for 3D data fields.*
2. Isolate a subset of the data *The ability to zoom in and clip a subregion of the data set is important for analyzing the data as a series of smaller pieces. Allowing the user to simplify the scene by subsetting the data makes it possible to tag the subset with additional information without incurring excessive clutter.*
3. Picking/Selecting data values *Being able to pick a data value or small set of values is an important feature. The user can query a point, getting its exact value, without cluttering the entire scene with unnecessary details.*

16.2 Volume Visualization with Texture

Volume rendering is a useful technique for visualizing three dimensional arrays of sampled data. Examples of sampled 3D data can range from computational fluid dynamics, medical data from CAT or MRI scanners, seismic data, or any volumetric information where geometric surfaces are difficult to generate or unavailable. Volume visualization provides a way to see through the data, revealing complex 3D relationships.

There are a number of approaches for visualization of volume data. Many of them use data analysis techniques to find the contour surfaces inside the volume of interest, then render the resulting geometry with transparency.

The 3D texture approach is a direct data visualization technique, using 2D or 3D textured data slices, combined using a blending operator [23]. The approach described here is equivalent to ray casting [44] and produces the same results. Unlike ray casting, where each image pixel is built up ray by ray, this approach takes advantage of spatial coherence. The 3D texture is used as a voxel cache, processing all rays simultaneously, one 2D layer at a time. Since an entire 2D slice of the voxels are "cast" at one time, the resulting algorithm is much faster with hardware-accelerated texture than ray casting.

This section is divided into two approaches, one using 2D textures, the other using a 3D texture. Although the 3D texture approach is simpler and yields superior results overall, 3D textures are currently still an `EXT` extension in OpenGL and are not universally available like 2D textures. 3D texturing will be available as part of OpenGL 1.2, so both methods [23] are described here.

16.2.1 Overview of the Technique

The technique for visualizing volume data is composed of two parts. First the texture data is sampled with planes parallel to the viewport and stacked along the direction of view. These planes are rendered as polygons, clipped to the limits of the texture volume. These clipped polygons are textured with the volume data, and the resulting images are blended together, from back to front, towards the viewing position. As each polygon is rendered, its pixel values are blended into the framebuffer to provide the appropriate transparency effect. See Figure 87.

If the OpenGL implementation doesn't support 3D textures, a more limited version of the technique can be used, where 3 sets of 2D textures are created, one set for each major plane of the volume data. The process then proceeds as with the 3D case, except that the slices are constrained to be parallel to one of the three 2D texture sets.

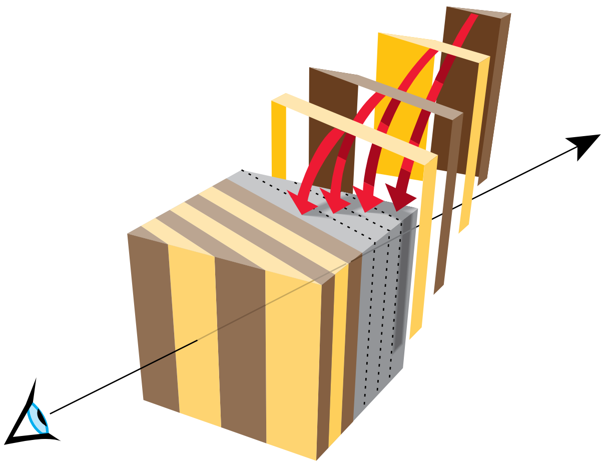


Figure 87. Slicing a 3D Texture to Render Volume

Close-up views of the volume cause sampling errors to occur at texels that are far from the line of sight into the data. To correct this problem, use a series of concentric tessellated spheres centered around the eye point, rather than a single flat polygon, to generate each textured "slice" of the data. As with flat slices, the spherical shells should be clipped to the data volume, and each textured shell blended from back to front. See Figure 88.

16.2.2 3D Texture Volume Rendering

Using 3D textures for volume rendering is the most desirable method. The slices can be oriented perpendicular to the viewer's line of sight, and creating spherical slices for close-up views doesn't lead to sampling errors.

Here are the steps for rendering a volume using 3D textures:

1. Load the volume data into a 3D texture. This is done once for a particular data volume.

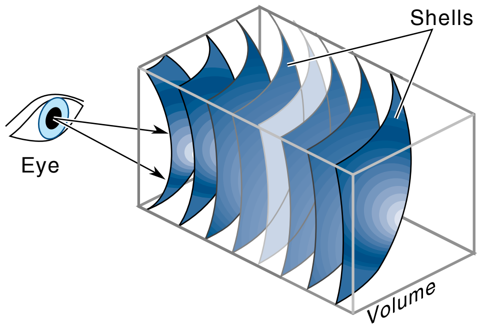


Figure 88. Slicing a 3D Texture with Spherical Shells

2. Choose the number of slices, based on the criteria in Section 16.2.5. Usually this matches the texel dimensions of the volume data cube.
3. Find the desired viewpoint and view direction.
4. Compute a series of polygons that cut through the data perpendicular to the direction of view. Use texture coordinate generation to texture the slice properly with respect to the 3D texture data.
5. Use the texture transform matrix to set the desired orientation of the textured images on the slices.
6. Render each slice as a textured polygon, from back to front. A blend operation is performed at each slice; the type of blend depends on the desired effect. See the blend equation descriptions in Section 16.2.4 for details.
7. As the viewpoint and direction of view changes, recompute the data slice positions and update the texture transformation matrix as necessary.

16.2.3 2D Texture Volume Rendering

Volume rendering with 2D textures is more complex and does not provide as good results as 3D textures, but can be used on any OpenGL implementation.

The problem with 2D textures is that the data slice polygons can't always be perpendicular to the view direction. Three sets of 2D texture maps are created, each set perpendicular to one of the major axes of the data volume. These texture sets are created from adjacent 2D slices of the original 3D volume data along a major axis. The data slice polygons must be aligned with whichever set of 2D texture maps is most parallel to it. In the worst case, the data slices are canted 45 degrees from the view direction.

The more edge-on the slices are to the eye, the worse the data sampling is. In the extreme case of an edge-on slice, the textured values on the slices aren't blended at all. At each edge pixel, only one sample is visible, from the line of texel values crossing the polygon slice. All the other values are obscured.

For the same reason, sampling the texel data as spherical shells to avoid aliasing when doing close-ups of the volume data, isn't practical with 2D textures.

Here are the steps for rendering a volume using 2D textures:

1. Generate the three sets of 2D textures from the volume data. Each set of 2D textures is oriented perpendicular to one of volume's major axes. This processing is done once for a particular data volume.
2. Choose the number of slices, based on the criteria in Section 16.2.5. Usually this matches the texel dimensions of the volume data cube.
3. Find the desired viewpoint and view direction.
4. Find the set of 2D textures most perpendicular to the direction of view. Generate data slice polygons parallel to the 2D texture set chosen. Use texture coordinate generation to texture each slice properly with respect to its corresponding 2D texture in the texture set.
5. Use the texture transform matrix to set the desired orientation of the textured images on the slices.
6. Render each slice as a textured polygon, from back to front. A blend operation is performed at each slice; the type of blend depends on the desired effect. See the blend equation descriptions in Section 16.2.4 for details.
7. As the viewpoint and direction of view changes, recompute the data slice positions and update the texture transformation matrix as necessary. Always orient the data slices to the 2D texture set that is most closely aligned with it.

16.2.4 Blending Operators

There a number of common blending functions used in volume visualization. They are described below.

Over The *over* operator [79] is the most common way to blend for volume visualization. Volumes blended with the over operator approximate the flow of light through a colored, transparent material. The transparency of each point in the material is determined by the value of the texel's alpha channel. Texels with higher alpha values tend to obscure texels behind them, and stand out through the obscuring texels in front of them.

The over operator can be implemented in OpenGL by setting the blend function to perform the over operation:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

Attenuate The *attenuate* operator simulates an X-ray of the material. With attenuate, the texel's alpha appears to attenuate light shining through the material along the view direction towards the viewer. The texel alpha channel models material density. The final brightness at each pixel is attenuated by the total texel density along the direction of view.

Attenuation can be implemented with OpenGL by scaling each element by the number of slices, then summing the results. This can be done by combination of the appropriate blend function and blend color:

```
glBlendFunc(GL_CONSTANT_ALPHA_EXT, GL_ONE)
glBlendColorEXT(1.f, 1.f, 1.f, 1.f/number_of_slices)
```

Maximum Intensity Projection Maximum Intensity Projection, or MIP, is used in medical imaging to visualize blood flow. MIP finds the brightest texel alpha from all the texture slices at each pixel location. MIP is a contrast enhancing operator; structures with higher alpha values tend to stand out against the surrounding data.

MIP can be implemented with OpenGL using the blend minmax extension:

```
glBlendEquationEXT(GL_MAX_EXT)
```

Under Volume slices rendered front to back with the *under* operator give the same result as the over operator blending slices from back to front. Unfortunately, OpenGL doesn't have an exact equivalent for the under operator, although using `glBlendFunc(GL_ONE_MINUS_DST, GL_DST)` is a good approximation. Use the over operator and back to front rendering for best results. See Section 8.1 for more details.

16.2.5 Sampling Frequency

There are a number of factors to consider when choosing the number of slices (data polygons) to use when rendering your volume:

Performance It's often convenient to have separate "interactive" and "detail" modes for viewing volumes. The interactive mode can render the volume with a smaller number of slices, improving the interactivity at the expense of image quality. Detail mode – rendering with more slices – can be invoked when the volume being manipulated slows or stops.

Cubical Voxels The data slice spacing should be chosen so that the texture sampling rate from slice to slice is equal to the texture sampling rate within each slice. Uniform sampling rate treats 3D texture texels as cubical voxels, which minimizes resampling artifacts.

For a cubical data volume, the number of slices through the volume should roughly match the resolution in texels of the slices. When the viewing direction is not along a major axis, the number of sample texels changes from plane to plane. Choosing the number of texels along each side is usually a good approximation.

Non-linear blending The over operator is not linear, so adding more slices doesn't just make the image more detailed. It also increases the overall attenuation, making it harder to see density details at the "back" of the volume. Strictly speaking, if you change the number of slices used to render the volume, the alpha values of the data should be rescaled. There is only one correct sample spacing for a given data set's alpha values. Generally, it doesn't buy you anything to have more slices than you have voxels in your 3D data.

Perspective When viewing a volume in perspective, the density of slices should increase with distance from the viewer. The data in the back of the volume should appear denser as a result of perspective distortion. If the volume isn't being viewed in perspective, then uniformly spaced data slices are usually the best approach.

Flat vs. Spherical Slices If you are using spherical slices to get good close-ups of the data, then the slice spacing should be handled in the same way as for flat slices. The spheres making up the slices should be tessellated finely enough to avoid concentric shells from touching each other.

2D vs. 3D Textures 3D textures can sample the data in the s, T, or r directions freely. 2D textures are constrained to s and t. 2D texture slices correspond exactly to texel slices of the volume data. To create a slice at an arbitrary point would require resampling the volume data.

Theoretically, the minimum data slice spacing is computed by finding the longest ray cast through the volume in the view direction, transforming the texel values found along that ray using the transfer function (if there is one), then finding the highest frequency component of the transformed texels, and using double that number for the minimum number of data slices for that view direction.

This can lead to a large number of slices. For a data cube 512 texels on a side, the worst case would be at least $1024\sqrt{3}$ slices, or about 1774 slices. In practice, however, the volume data tends to be bandwidth limited; and in many cases choosing the number of data slices to be equal to the volume's dimensions, measured in texels, works well. In this example, you may get satisfactory results with 512 slices, rather than 1774. If the data is very blurry, or image quality is not paramount (for example, in "interactive mode"), this value could be reduced by a factor of two or four.

16.2.6 Shrinking the Volume Image

For best visual quality, render the volume image so that the size of a texel is about the size of a pixel. Besides making it easier to see density details in the image, larger images avoid the problems associated with undersampling a minified volume.

Reducing the volume size will cause the texel data to be sampled to a smaller area. Since the over operator is non-linear, the shrunken data will interact with it to yield an image that is different, not just smaller. The minified image will have density artifacts that are not in the original volume data.

If a smaller image is desired, first render the image full size in the desired orientation, then shrink the resulting 2D image.

16.2.7 Virtualizing Texture Memory

Volume data doesn't have to be limited to the maximum size of 3D texture memory. The visualization technique can be virtualized by dividing the data volume into a set of smaller "bricks". Each brick is loaded into texture memory, then data slices are textured and blended from the brick as usual. The processing of bricks themselves is ordered from back to front relative to the viewer. The process is repeated with each brick in the volume until the entire volume has been processed.

To avoid sampling errors at the edges, data slice texture coordinates should be adjusted so they don't use the surface texels of any brick. The bricks themselves are oriented so that they overlap by one volume texel with their immediate neighbors. This allows the results of rendering each brick to combine seamlessly. For more information on paging textures, see Section 6.8.

16.2.8 Mixing Volumetric and Geometric Objects

In many applications it is useful to display both geometric primitives and volumetric data sets in the same scene. For example, medical data can be rendered volumetrically, with a polygonal prosthesis placed inside it. The embedded geometry may be opaque or transparent.

The Opaque geometric objects are rendered first using depth buffering. The volumetric data slice polygons are then drawn, with depth testing still enabled. Depth buffer updating should be masked off if the slice polygons are being rendered from front to back (for most volumetric operators, data slices are rendered back to front). With depth testing enabled, the pixels of volume planes behind the object aren't rendered, while the planes in front of the object blend it in. The blending of the planes in front of the object gradually obscure it, making it appear embedded in the volume data.

If the object itself should be transparent, it must be rendered along with the data slice polygons a slice at a time. The object is chopped into slabs using user defined clipping planes.The slab thickness corresponds to the spacing between volume data slices. Each slab of object corresponds to one of the data slices. Each slice of the object is rendered and blended with its corresponding data slice polygon, as the polygons are rendered back to front.

16.2.9 Transfer Functions

Different alpha values in volumetric data often correspond to different materials in the volume being rendered. To help analyze the volume data, a non-linear transfer function can be applied to the texels, highlighting particular classes of volume data. This transformation function can be applied through one of OpenGL's lookup tables. The `SGI_texture_color_table` extension applies a lookup table to texels values during texturing, after the texel value is filtered.

Since filtering adjusts the texel component values, a more accurate method is to apply the lookup table to the texel values before the textures are filtered. If the `EXT_color_table` table extension is available, then a colortable in the pixel path can be used to process the texel values while the texture is loaded. If lookup tables aren't available, the processing can be done to the volume data by the application, before loading the texture.

If the paletted texture extension (`EXT_paletted_texture`) is available and the 3D texture can be stored simply as color table indices, it is possible to rapidly change the resulting texel component values by changing the color table.

16.2.10 Volume Cutting Planes

Additional surfaces can be created on the volume with user defined clipping planes. A clipping plane can be used to cut through the volume, exposing a new surface. This technique can help expose the volume's internal structure. The rendering technique is the same, with the addition of one or more clipping planes defined while rendering and blending the data slice polygons.

16.2.11 Shading the Volume

In addition to visualizing the voxel data, the data can be lit and shaded. Since there are no explicit surfaces in the data, lighting is computed per volume texel.

The direct approach to shading is to do it on the host. The volumetric data can be processed to find the gradient at each voxel. Then the dot product between the gradient vector, now used as a normal, and the light is computed, and the results saved as 3D data. The volumetric data now contains the intensity at each point in the data, instead of data density. Specular intensity can be computed the same way, and combined so that each texel contains the total light intensity at every sample point in the volume. This processed data can then be visualized in the manner described previously.

The problem with this technique is that a change of light source (or viewer position, if specular lighting is desired) requires that the data volume be reprocessed. A more flexible approach is to save the components of the gradient vectors as color components in the 3D texture. Then the lighting can be done while the data is being visualized. One way to do this is to transform the texel data using the color matrix extension. The light direction can be processed to form a matrix that when multiplied by the texture color components (now containing the components of the normal at that point), will produce the dot product of the two. The color matrix is part of the pixel path, so this processing can be done when the texture is being loaded. Now the 3D texture contains lighting intensities as before, but the dot product calculations are done in the pixel pipeline, not in the host.

The data's gradient vectors could also be computed interactively, as an extension of the texture bump-mapping technique described in Section 10.6. Each data slice polygon is treated as a surface polygon to be bump-mapped. Since the texture data must be shifted and subtracted, then blended with the shaded polygon to generate the lit slice before blending, the process of generating lit slices must be processed separately from the blending of slices to create the volume image.

16.2.12 Warped Volumes

The data volume can be warped by non-linearly shifting the texture coordinates of the data slices. For more warping control, tessellate the vertices to provide more vertex locations to perturb the texture coordinate values. Among other things, very high quality atmospheric effects, such as smoke, can be produced with this technique.

16.3 Vector Field Visualization

Visualizing vector fields is a difficult problem. Whereas scalar fields have a scalar value at each sample point, vector fields have an n-component vector (usually 2 or 3 components) at each point. Vector fields occur in applications such as computational fluid dynamics and typically represent the flow of a gas or liquid. Visualization of the field provides a way to better observe and understand the flow patterns.

Vector field visualization techniques can be grouped into 3 general classes:

Icon Based Icon based techniques render a 3D geometric object (cone, arrow, etc) at each sample point with the geometry aligned with the vector direction at that point. Other attributes such as object size or color can be used to encode a scalar quantity such as the magnitude of the vector at each sample point.

Particle Tracing Based Particle tracing based techniques use point-shaped geometry to trace out paths through the vector field. Portions of the field are seeded with particles and paths are traced through the field following the vector field samples. The positions of the particles along their respective paths are animated over time to convey a sense of flow through the field. A variation on the techniques is to trace out the paths and display each path as a *stream line* using lines, ribbons or tubed shaped geometry.

Texture Based Texture based techniques use image processing algorithms to trace out many paths through the field simultaneously. These techniques work very well for 2D fields or cross sections of 3D fields resulting in dense and detailed stream line images.

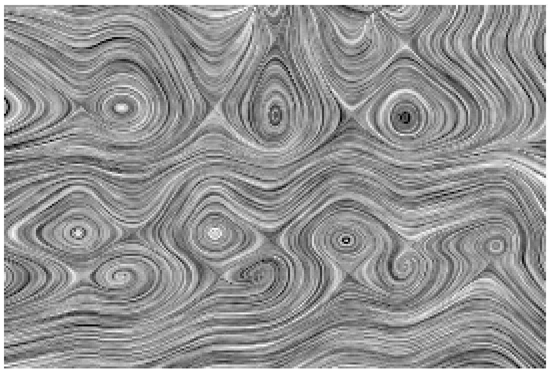


Figure 89. Line Integral Convolution

16.4 Line Integral Convolution (LIC) with Texture

Line integral convolution is texture based technique for visualizing vector fields and has the advantage of being able to visualize large and detailed vector fields in a reasonable display area.

Line integral convolution involves selectively blurring a reference image as a function of the vector field to be displayed. The reference image can be anything, but to make the results clearer, is usually an spatially uncorrelated image (e.g., a noise image). The resulting image appears stretched and squished along the directions of the distorting vector field streamlines, visualizing the flow with a minimum of display resolution. Vortices, sources, sinks and other discontinuities are clear shown in the resulting image, and the viewer can get an immediate grasp of the flow fields "big picture".

In each case, you start with a vector field, sampled as a discrete grid of normalized vectors. You also need an image that is non-uniform and spatially uncorrelated, so correlations you apply to it will be more obvious. The goal is to process the image with the vector field, using line integral convolution, so you can visualize it. Note that in this technique, you will concentrate on the direction of the flow field, not its velocity; this is why the vector values at each gridpoint are normalized.

The processed image can be calculated directly using a special convolution technique. A representative set of vector values on the vector grid are chosen. Special convolution kernels are created shaped like the local stream line at that vector by tracing local field flow forwards and backwards some user-defined distance. The resulting curve is used as a convolution kernel to convolve the underlying image. This process is repeated over the entire image using a sampling of the vectors in the vector field.

Mathematically, for each location p in the input vector field, a parametric curve $P(p, s)$ is generated which passes through the location and follows the vector field for some distance in either direction. To create an output pixel $F'(p)$, a weighted sum of the values of the input image F along the curve is computed. The weighting function is $k(x)$. Thus the continuous form of the equation is:

$$F'(p) = \frac{\int_{-L}^{L} F(P(p,s))k(s)ds}{\int_{-L}^{L} k(s)ds}$$

To discretize the equation, use values $P_{0..l}$ along the curve $P(p, s)$:

$$F'(p) = \frac{\sum_{i=0}^{l} F(P_i)h_i}{\sum_{i=0}^{l} h_i}$$

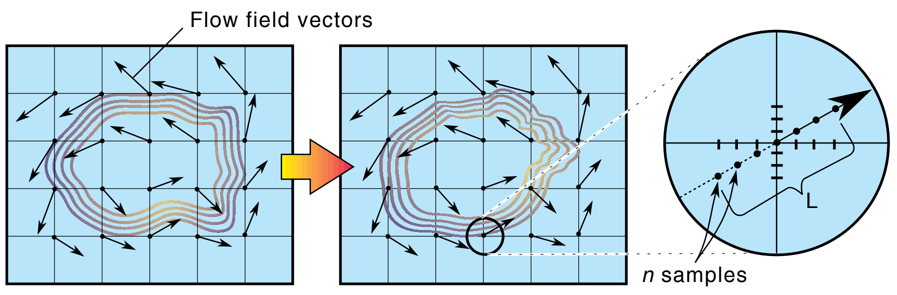


Figure 90. Line Integral Convolution with OpenGL

16.4.1 Sampling

How accurately the processed image represents the vector field depends on how accurately the line convolution kernels follow the flow fields stream lines. Since the convolution kernels are only discretely sampling a continuous flow field, they are inaccurate in general. Areas of flow that are changing slowly will be represented well, but rapidly changing regions of the flow field (such as the center of vortices and other singularities) will be incorrectly described or missed altogether.

There are various ways of optimizing the sampling intervals to minimize this this problem, with different tradeoffs between computation time and resulting accuracy. The numerical analysis topics involved are beyond the scope of this document, and are well covered elsewhere [16, 61]. For our purposes, we'll use the simplest and least accurate method – a fixed spatial sampling interval.

16.4.2 Using OpenGL to Create Line Integral Convolution (LIC) Images

Instead of generating a series of custom convolution kernels and applying them to an image, you can use a texture mapping approach. This variant has the advantage that it's reasonably easy to implement and runs quickly, especially on systems with good texturing and accumulation buffer support, since it is parallelizing the convolution operations.

The concept is simple; a surface, tessellated into a mesh, is textured with an image to be processed. Each vertex on the surface has a texture coordinate associated with it. Instead of convolving the image with a series of streamline convolution kernels, the texture coordinates at each vertex are shifted parallel to flow field vector local to that vertex. This process, called advection, is done repeatedly in a series of displacements parallel to the flow vectors, with the resulting series distorted images combined using the accumulation buffer.

The texture coordinates at each grid location are displaced parallel to the local field vector in a fixed series of steps. The displacement is done both parallel and antiparallel to the field vector at the vertex. The amount of displacement for each step and the number of steps determines the accuracy and appearance of the line integral convolution. The application generally sets a global value describing the length of the displacement range for all of the texture coordinates on the surface; the number of displacements along that length is computed per vertex, as a function of the local field's curl.

16.4.3 Line Integral Convolution Procedure

Next, make some simplifying assumptions to make the procedure simple:

1. The supplied flow field vector grid matches the tessellated textured surface; there's a one-to-one correspondence between vector and vertex.
2. Set a fixed number of displacements (n) at each vertex.

These assumptions allow you to simply use the vector associated with each vertex on the tessellated surface when computing texture displacements. You can also simply calculate the displacements by parameterizing the vector and computing evenly spaced texture coordinate locations displaced along the vector direction, both forwards and backwards.

Given these assumptions, the procedure looks like this:

1. Update the texture coordinates at each vertex on the surface.
2. Render the surface using the noise texture and the displaced texture coordinates.
3. Accumulate the resulting image in the accumulation buffer, scaling by $1/n$.
4. Repeat the steps above n times, then return the accumulated image.
5. Perform histogram equalization or image scaling to maximize contrast.

16.4.4 Details

Since the most of the work goes into updating the texture coordinates, it makes sense to use vertex arrays to represent the textured surface. Using a vertex array provides two benefits; it simplifies the representation of the texture coordinates (they can be kept in a 2D array), and it potentially increases rendering performance since using `glDrawElements` has an index array that can eliminate the need for sending shared texture and vertex coordinates multiple times, and reduces function call overhead.

Scaling each accumulation uniformly is not optimal. The displacement of the texture coordinates is most accurate close to the grid vector; so each image contribution can be scaled as an inverse function of distance from from the vector. The farther the displacement from the original flow field vector, the less accurate the advection can potentially be, and the smaller accumulation scale factor is. Obviously more sophisticated algorithms can be implemented that adjust scale based on a computed, rather than assumed, accuracy. Any scaling algorithm should take into account the maximum and minimum possible color values after accumulating to avoid pixel color overflow or underflow.

In many implementations, the performance of this algorithm will be limited by the speed of the convolution operation. For some applications, a blend operation can be substituted with a loss of resolution accuracy; the scaling operation can be provided by changing the intensity of the base polygon. Watch out for overflow and underflow of the blended color values.

16.4.5 Maximizing Contrast

There are a couple of obvious methods to maximize the effects of the flow field being visualized, in particular, to contract the blurring tendency from the the random noise texels being blended together. One simple method is to scale and bias the image to maximize its contrast. The imaging subset makes this easy. Process the image by doing a pixel copy, turning on sink after the minmax operation. With the minimum and maximum values obtained, you can execute `glCopyPixels` again, setting scale and bias in the pixel pipeline to scale and bias the image.

Or you can do a full histogram equalization. Using the histogram feature, copy the image through the pixel pipeline, then process the resulting histogram to create a lookup table. The lookup table will balance the intensities into a linear ramp. Again use copypixels to remap the pixel intensity values. In detail:

1. `glEnable(GL_MIN_MAX)`
2. `glMinmax(GL_MIN_MAX, GL_LUMINANCE, GL_TRUE)`
3. `glCopyPixels` of LIC Image.
4. `glGetMinmax` to get minimum and maximum pixel values.
5. Compute a scale and bias value to get full 0 to 1 dynamic range.
6. `glDisable(GL_MIN_MAX)`
7. `glDisable(GL_MIN_MAX)`
8. `glPixelTransfer` to set scale and bias value.
9. `glCopyPixels` of LIC Image to rescale it.

16.4.6 Going Farther

The approach described here to generate line integral convolution images is very simplistic. More sophisticated algorithms will decouple the surface tessellation from the flow field grid, and more finely subdivide the tessellation surface where there rapidly changing flows to properly sample them. This subdivision algorithm should be backed with a rigorous sampling approach so that the results can can be trusted within given accuracy bounds. A subdivision algorithm must also recognize and handle various types of flow discontinuities.

This technique can easily be extended into three dimensions, using 3D textures. Volume visualization techniques, described in Section 16.2 in these notes, can be used to visualize the 3D LIC image.

16.5 Illuminated Stream Lines

Stream line based techniques rely on tracing a path through the field starting at some point in the field. Once the path has been traced, the path is represented using geometric primitives. When visualizing 3-dimensional fields, illumination and shading provide additional visual cues, particularly for dense collections of stream lines. One type of geometry that can be used is tube shaped geometry constructed from segments of cylinders following the path. In order to capture accurate shading information the radius of the cylinders needs to be finely tessellated resulting in a large polygon load when displaying a large number of stream lines.

Another possibility it to use line primitives since they can be rendered very efficiently and allow very large numbers of streamlines to be drawn. A disadvantage is that lines are rendered as *flat* geometry with a single normal at each end point so they result in much lower shading accuracy compared to using tessellated cylinders. In [92] an algorithm is described to approximate cylinder-like lighting using texture mapping.

The main idea behind the algorithm is to choose a normal vector that lies in the same plane as that formed by the tangent vector $\vec{T}$ and light vector $\vec{L}$. The diffuse and specular lighting contributions are then expressed in terms of the line's tangent vector and the light vector rather than a normal vector. For the diffuse term, $\vec{L} \cdot \vec{N} = \sqrt{1 - (\vec{L} \cdot \vec{T})^2}$. This allows the diffuse cosine term to be evaluated using the OpenGL texture matrix to compute $\vec{L} \cdot \vec{T}$, by specifying the components of the tangent vector in the texture coordinate and the components of the light vector in the texture matrix,

$$M = \begin{pmatrix} L_x & L_y & L_z & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

The resulting s texture coordinate is $t_s = (\vec{L} \cdot \vec{T} + 1)/2$ which has been biased to ensure that the computed coordinate value lies in the range $[0, 1]$. The s texture coordinate is then used to index a 1-dimensional texture storing the cosine function modulated by the material diffuse reflectance $f(t_s) = d_m\sqrt{1 - (2t_s - 1)^2}$.

Similarly the specular term $(\vec{V} \cdot \vec{R})^n$ can be expressed in terms of the tangent vector and viewing direction $\vec{V}$ as $((\vec{L} \cdot \vec{T})(\vec{V} \cdot \vec{T}) - \sqrt{1 - (\vec{L} \cdot \vec{T})^2}\sqrt{1 - (\vec{V} \cdot \vec{T})^2})^n$. Using the texture matrix,

$$M = \begin{pmatrix} L_x & L_y & L_z & 1 \\ V_x & V_y & V_z & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

to compute $t_t = (\vec{V} \cdot \vec{T} + 1)/2$, then $\vec{V} \cdot \vec{T} = 2t_t - 1$ and $t_s = (\vec{L} \cdot \vec{T} + 1)/2$ as before. Thus t_s and t_t can be used with a 2-dimensional texture to generate the specular term. Since the texture encodes the viewing direction $\vec{V}$, the texture map must be regenerated if the view direction changes (if the light moves with the view, as is the case for a headlight, then no recomputation is necessary). Finally, the specular and diffuse terms can be combined together with an ambient term into a single texture map to perform the entire lighting calculation in a single pass.

Since the texture map encodes the material reflectance coefficients, multiple texture maps are required for multiple materials. This need for multiple maps can be eliminated by sending the material color as the line color and storing only the light intensity in the texture map. The two can be combined using the GL_MODULATE texture environment function. Multiple maps or recomputations are required to support different combinations of diffuse and specular properties.

The illuminated lines can also be rendered using transparency techniques. This is useful for dense collections of stream lines. The opacity value is sent with the line color and the lines must be sorted from back to front to be rendered correctly as described in Section 12.

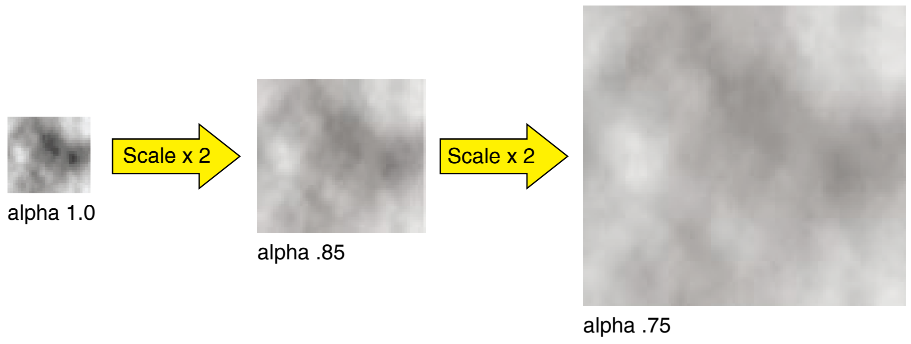


Figure 91. Dilating, Fading Smoke

17 Natural Phenomena

The are a large number of naturally occurring phenomena such as smoke, fire and clouds which are challenging to render at interactive rates with any semblance of realism. A common solution is to reduce the requirement for complex geometry by using textures. Many of the techniques use a combination of geometry and texture which vary as a function of time or other parameters such as distance from the viewer.

17.1 Smoke

Modeling smoke potentially requires some sophisticated physics, but surprisingly realistic images can be generated using fairly simple techniques. One such technique involves capturing a 2D cross section or image of a puff of smoke with both luminance and alpha channels for the image. The image can then be texture mapped onto a quadrilateral and blended into the scene. The billboard techniques outlined in Section 6.10 can be used to ensure that the image is transformed to face the user. Using a `GL_MODULATE` texture environment, the color and alpha value of the quadrilateral can be used to control the color and transparency of the smoke in order to simulate different types of smoke. For example, smoke from an oil fire would be dark and opaque, whereas steam from a flare stack would be much lighter in color.

The size, position, orientation, and opacity of the quadrilateral can be varied as a function of time to simulate the puff of smoke enlarging, drifting and dissipating over time.

More realistic effects can be achieved using volumetric techniques. Instead of a 2D image, a 3D volumetric image of smoke is rendered using the algorithms described in Section 16.2. Again, dynamics can be simulated by varying the position, size and transparency of the volume. More complex dynamics can be simulated by applying local distortions or deformations to the texture coordinates of the volume lattice rather than simply applying uniform transformations. The volumetric shading technique described in Section 16.2.11 can be used to illuminate the smoke.

There are many procedural techniques which can be used to synthesize both 2D and 3D textures [25].

17.2 Vapor Trails

Vapor trails emanating from a jet or a missile can be rendered using methods similar to the painting technique described in Section 8.3. A circular, wispy 2D image such as that used in the preceding section is used to generate the vapor pattern over some unit interval by rendering it as a billboard. A texture image consisting only of alpha

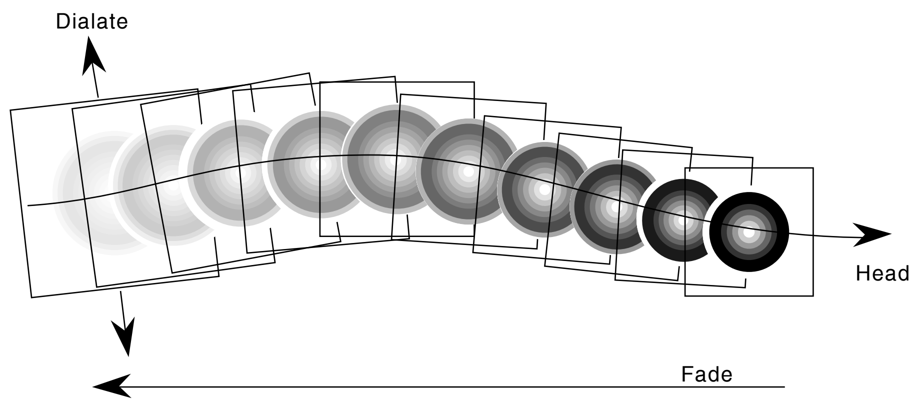


Figure 92. Vapor Trail

values is used to modulate the alpha values of a white billboard polygon. The trajectory of the airborne object is painted using multiple overlapping copies of the billboard as shown in Figure 92. Over time the individual billboards gradually enlarge and fade. The program for rendering a trail is largely an exercise in maintaining an active list of the position, orientation and time since creation for each billboard used to paint the trail. As each billboard polygon exceeds a threshold transparency value it can be discarded from the list.

17.3 Fire

The simplest techniques for rendering fire involve applying static images and movie loops as textures to billboards.

A static image of fire can be constructed from a noise texture; Section 6.20.2 describes how to make a noise texture using OpenGL. The weights for different frequency components should be chosen to reflect the spectral structure of fire, and turbulence can also be incorporated effectively into the texture. The texture is mapped to a billboard polygon. Several such textures, composited together, can create the appearance of multiple layers of intermingling flames. Finally, the texture coordinates may be distorted vertically to simulate the effect of flames rising and horizontally to mimic the effect of winds.

A sequence of fire textures can be played as an animation. The abrupt manner in which fire moves and changes intensity can be modeled using the same turbulence techniques used to create the fire texture itself. The speed of the animation playback, as well as the distortion applied to the texture coordinates of the billboard, might be controlled using a turbulent noise function. To create the animation a series of texture objects is created, each one containing one image from the fire sequence. During playback the set of texture objects is sequenced through, one each frame, mapping the current texture to a quadrilateral using a modulate texture environment.

17.4 Explosions

Explosion effects can be rendered by combining the techniques for smoke, vapor, and fire. A static image of a fireball is drawn centered in the middle of the explosion and dilated and faded over some time period. At the same time, the vapor and smoke rendering techniques are combined to cause a smoke trail to rise from the center of the explosion. To make the explosion appear more realistic, the geometry for fragments of objects are added to the scene with their own animated trajectories.

17.5 Clouds

Clouds, like smoke, have an amorphous structure without well defined surfaces and boundaries. In recent times, computationally intensive physical modeling techniques have given way to simplified mathematical models which are both computationally tractable and aesthetically pleasing [31, 25].

The main idea behind these techniques involves generating a realistic 2D or 3D texture function t using a fractal or spectral based function. Gardner suggests a Fourier-like sum of sine waves with phase shifts

$$t(x, y) = k \sum_{i=1}^{n} (c_i \sin(fx_i x + px_i) + t_0) \sum_{i=1}^{n} (c_i \sin(fy_i y + py_i) + t_0)$$

with the relationships

$$\begin{aligned} fx_{i+1} &= 2fx_i \\ fy_{i+1} &= 2fy_i \\ c_{i+1} &= .707c_i \\ px_i &= \frac{\pi}{2}\sin(fy_{i-1}y), i > 1 \\ py_i &= \frac{\pi}{2}\sin(fx_{i-1}x), i > 1 \end{aligned}$$

Care must be taken using this technique to choose values to avoid a regular pattern in the texture. Alternatively, texture generation techniques described in Section 6.20.2 can be used.

A stochastic method, based on work by Fournier and Miller [29, 63], uses a midpoint displacement technique called Diamond-Square for generating a set of random values on a uniform grid. These generated values are interpreted as opacity values and correspond to the cloud density at a given point. The algorithm is iterative and during each iteration two steps are executed. The first, the *diamond* step takes four corners of a square and produces a new value at the center of the square by averaging the values at the four corners and adding a random number in the range $[-1, 1]$. The second step, the *square* step, consists of taking the corners of the four diamonds that were generated in the diamond step (they share the center point of the diamond step) and generating a new center value for each diamond by averaging its four corners and adding a random number in the range $[-1, 1]$. During the square step, attention must be paid to diamonds at the edges of the grid as they will wrap around to the opposite side of the grid. During each iteration the number of squares processed is increased by a factor of four. To produce smooth variations in the generated values, the range of the random value added during the generation of center points is reduced by some fraction for each iteration.

Seed values for the first few iterations of the algorithm may be used to control the overall shape of the cloud.

Any of these techniques can be used to produce a 2D texture which can be used to render a *cloud layer*. A cloud layer is simulated by drawing a large textured polygon in the sky at a fixed altitude. A luminance cloud texture is used to blend a white constant texture environment color into a blue sky polygon.

Some of the dynamic aspects of clouds can be simulated by vary parameters over time. Cloud development can be simulated by scaling and biasing the luminance values in the texture. Drifting can be simulated by moving the texture pattern across the sky, i.e., transforming the texture coordinates. Ground fog can be simulated by drawing the thin cloud layer between the viewer and ground rather than the viewer and the sky.

Gardner also suggests using ellipsoids to simulate 3D cloud structures. The texture data is generated using a 3-dimensional extension of the Fourier synthesis method outlined above and the textures are applied with increasing transparency near the boundary of the ellipsoid. These 3D textures can also be combined with the volume rendering techniques described in Section 16.2 to produce 3D cloud images. In order to improve the performance of the rendering, the full volume rendering algorithm need not be used. In particular, the cloud may be assumed to be elliptical and opaque at the center. Therefore, the interior of the cloud can be drawn as a polygonal shell and the outer edges of the cloud using the volume rendering techniques.

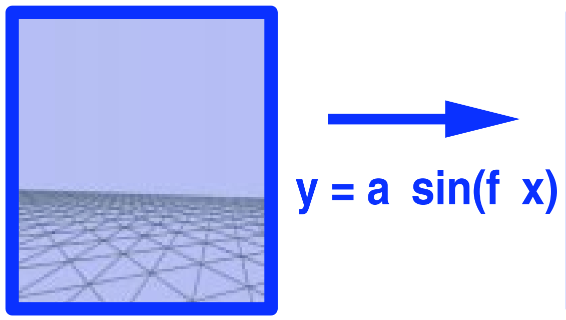


Figure 93. Water Modeled as a Height Field

17.6 Water

A large body of research has been done into modeling, shading, and reproducing optical effects of water [100, 75, 30], yet most methods still present a large computation burden to achieve a realistic image. Nevertheless, it is possible to borrow from these approaches and achieve modest results while retaining interactive performance [54, 25].

The dynamics of wind and waves can be simulated using procedural models and rendered using meshes or height fields. The geometry is textured using simple procedural texture images. Multipass rendering techniques can be used to layer additional effects such as surf. Environment mapping can be used to simulate reflections from the surface. Specular illumination using environment mapping can be combined with the Fresnel reflection model from Section 10.4 to create a more physically accurate lighting model. The bump mapping technique from Section 10.6 can be used to create the illusion of ripples without modeling them in the geometry. The bump map can be animated as part of the simulation to animate the ripples. The combination of reflection mapping and a dynamic model for ripples provides a visually compelling image. Alternatively, synthetic perturbations to the texture coordinates as outlined in Section 6.21.7 can also be used.

Small swells can be modeled using a texture mapped height field. The height of the vertices can be modulated with a sinusoid to simulate simple wave patterns as showing in Figure 93. The frequency and amplitude of the waves can be varied to achieve different effects. The phase of the sinusoid can be varied over time to create wave motion.

Optical effects such as caustics can be approximated using parts of the OpenGL pipeline as described by Nishita and Nakamae [70] but interactive frame rates are not likely to be achieved. Instead such effects can be faked using textures to modulate the intensity of any geometry that lies below the surface. Other below-surface effects can also be simulated. Movements of the water (surge) can be simulated by perturbing the vertex coordinates of submerged objects, again using sinusoids. Blueish-green fog can be used to simulate light attenuation in water.

17.7 Light Points

OpenGL has direct support for rendering both aliased and antialiased points, but these simple facilities are usually insufficient for simulating small light sources, such as stars, beacons, runway lights, etc. In particular, the size of OpenGL points is not affected by perspective projections. To render more realistic looking small light sources it is necessary to change some combination of the size and brightness of the source as a function of distance from the eye.

The brightness attenuation a as a function of distance, d, can be approximated by using the same equation used in the OpenGL lighting equation

$$\frac{1}{k_c + k_l d + k_q d^2}$$

Attenuation can be achieved by modulating the point size by the square root of the attenuation

$$size_{effective} = size \times \sqrt{a}$$

As the point size approaches the size of a single pixel the resolution of the raster display system will cause artifacts. To avoid this problem the point can be made semi-transparent once it crosses a particular size threshold. The alpha value is proportional to the ratio of the point area determined from the size attenuation computation to the area of the point being rendered

$$alpha = \left(\frac{size_{effective}}{size_{threshold}}\right)^2$$

More complex behavior such as defocusing, perspective distortion and directionality of light sources can be achieved by using an image of the light lobe as a texture map combined with billboarding to keep the light lobe oriented towards the viewer. An advantage of using texture mapping is that the quadrilateral or other geometry that the texture is applied to is automatically scaled by the perspective projection so rendering the correct size is less of an issue. To effectively simulate distance attenuation it may, however be necessary to select different texture patterns according to distance from the eye.

17.8 Other Atmospheric Effects

OpenGL provides a primitive capability for rendering atmospheric effects such as fog, mist and haze. It is useful to simulate the affects of atmospheric effects on visibility to increase realism, and it allows the database designer to cover up a multitude of sins such as "dropping" polygons near the far clipping plane in order to sustain a fixed frame rate.

OpenGL implements fogging by blending the fog color with the incoming fragments using a fog blending factor, f,

$$C = fC_{in} + (1 - f)C_{fog}$$

This blending factor is computed using one of three equations: exponential (`GL_EXP`), exponential-squared (`GL_EXP2`), and linear (`GL_LINEAR`)

$$\begin{aligned} f &= e^{-(density \cdot z)} \\ f &= e^{-(density \cdot z)^2} \\ f &= \frac{end - z}{end - start} \end{aligned}$$

where z is the eye-coordinate distance between the viewpoint and the fragment center.

Linear fog is frequently used to implement intensity depth-cuing in which objects closer to the viewer are drawn at higher intensity [27]. The effect of intensity as a function of distance is achieved by blending the incoming fragments with a black fog color.

The exponential fog equation has some physical basis. It is the result of integrating a uniform attenuation between the object and the viewer. The exponential-squared function includes the attenuation for reflected light which has passed through the attenuation layer twice, once for the incident path and again for the reflected path. The exponential and exponential-squared functions can be used to represent a number of atmospheric effects using different combinations of fog colors and density values. Since OpenGL does not fog the pixel values during a clear operation, the value of f at the far plane, far,

$$f_{far} = e^{-(density \cdot far)}$$

can be used to determine the color to which to clear the background

$$C_{bg} = f_{far}C_{in} + (1 - f_{far})C_{fog}$$

where C_{in} is the color to which the background would be cleared without fog enabled.

As mentioned earlier, the obscured visibility of objects near the far plane can be exploited to overcome various problems such as drawing time overruns, level-of-detail transitions, and database paging. However, in practice it has been found that the exponential function does not attenuate distant fragments rapidly enough, so exponential-squared fog can be used to achieve a sharper fall-off in visibility. Some vendors have gone a step further and provided more control over the fog function by allowing applications to control the fog value through a spline curve.

There are other problems that OpenGL's primitive fog model does not address. For example, emissive geometry such as the light points described above should be attenuated less severely than non-emissive geometry. This effect can be approximated by precompensating the color values for emissive geometry, or reducing the fog density when emissive geometry is drawn. Neither of these solutions is completely satisfactory since colors values are clamped to 1.0 in OpenGL, limiting the amount of precompensation that can be done. Many OpenGL implementations use lookup table methods to efficiently compute the fog function, so changes to the fog density may result in expensive table recomputations. To overcome this problem some vendors have provided a mechanism to bias the eye-coordinate distance, avoiding the need to recompute the fog lookup table.

If OpenGL fog processing is bypassed it is possible to do more sophisticated atmospheric effects using multipass techniques. The OpenGL fog computation can be thought of as simple table lookup using the eye-coordinate distance. The result is used as a blend factor for blending between the fragment color and fog color. A similar operation can be implemented using `glTexGen` to generate the eye-coordinate distance for each fragment and a 1D texture for the fog function. Using a specially constructed 2D or 3D texture and a more sophisticated, texture coordinate generation function, it is possible to compute more complex fog functions incorporating parameters such as altitude and eye-coordinate distance.

17.9 Particle Systems

Some objects are difficult to represent as a set of surface primitives, even taking advantage of transparency and texture mapping techniques. These include objects that have poorly defined or dynamic topologies, or have no solid surface. Natural phenomena that meet this criteria include smoke, clouds, fire, water, etc.

Particle systems can be used to represent these objects. A particle system is a large set of simple primitive objects which are processed as a group to represent an object. The characteristics of these objects, such as size, position, color, and the lifetime of the particle itself, can be changed dynamically. If these parameters of the particles are coordinated, the collection of particles can represent an object.

17.9.1 Representing Particles

Since you would like to use a lot of particles to create more realistic objects, you would like to render them as cheaply as possible. One good candidate primitive is an OpenGL point. Unaliased single points of default size are rendered as single fragments. They can be thought of as very small screen aligned rectangular billboards, since they are always oriented towards the viewer.

It is important to pass points to the graphics hardware as efficiently as possible. Display lists are very efficient, but since the characteristics of the points are usually changing from frame to frame, vertex arrays would be a better choice. Vertex arrays avoid the overhead of multiple function calls per vertex, and have an additional advantage; the primitive data is organized in array form. This is useful since some or all of the point characteristics must be updated by the program each frame. It is important that this be done efficiently, or the updating can become the bottleneck, starving the graphics hardware.

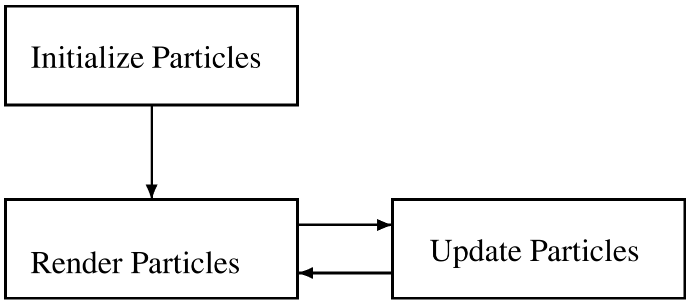


Figure 94. Particle System Block Diagram

Index	X, Y, Z	R, G, B, A	Vx, Vy, Vz	Lifetime Count
0				
1				
2				
3				
...				

A particle system program has these basic components:

Particles in particle systems can be organized in tables, indexed by the particle, containing particle characteristics to be updated each frame. This representation works well with vertex array representation, since the tables can be used directly to render the updated particles.

Interleaved or non-interleaved vertex arrays can be used, depending on the complexity of the particle system parameters. Parameters directly used for rendering, such as x, y, z position can be intermixed in the table with non-rendering parameters, such as current velocity. Vertex array strides can be adjusted to intermix these two types of information, or they can be kept separated. Since particle update performance is important, particle tables may have many non-rendering values to support incremental update algorithms.

When choosing a vertex array representation, keep in mind that OpenGL implementations often have higher performance using interleaved arrays that are densely packed. We recommend using `glInterleavedArrays` when possible. Of course, the data structure may have be adjusted to optimize for either rendering speed or particle update performance, depending on which part of the system is the performance bottleneck.

17.9.2 Particle Sizes

If particles are very small, or the particles are clustered tightly together some distance from the viewer, good effects are possible with particles of a single size. If the particles are moving a large distance towards or away from the viewer, a constant sized particle may appear unrealistic. Particles of changing sizes can lead to performance penalties. Changing point size can be a costly operation in OpenGL. Whenever possible, sort and group the particles by size when rendering to minimize the number of `glPointSize` calls. Sorting overhead can be minimized in many cases by using an incremental sorting algorithm, since points generally move only a small distance from frame to frame.

If the `GL_EXT_point_parameters` extension is available, you can use `glPointParameterfEXT` and `glPointParameterfvEXT` to set parameters that control point size as a function of distance from the viewer. This extension should be carefully benchmarked to see if your implementation can handle a set points with unsorted distance values efficiently. If not, then the points should still be sorted (or perhaps just partially sorted) to increase rendering efficiency.

Often sorting can be minimized by quantizing point sizes to a few distinct values. Groups of points within a given bounding volumes can be all set to an average size appropriate for that volume. As before, the effectiveness of quantizing particle size will depend on the behavior of particles in a particular system.

17.9.3 Large and Small Points

If the particle size is increased from the default, the rectangular nature of the point representation may become too apparent. Point antialiasing can be used to render the points as circles rather than squares. Benchmark the performance of antialiased points of various sizes on your system to determine the overhead of using this feature. Be sure to also take into account the fact that you will have to use alpha blending to make point antialiasing work.

If a particle must appear smaller than a single pixel, its alpha value can be reduced to make it more transparent (remember to enable blending), simulating the brightness of a smaller particle. Another technique that is faster but may not look as good is to reduce the intensity of the particle's color instead of its alpha. See Section 17.7 for more information.

17.9.4 Antialiasing

Antialiasing particles, both spatially and temporally, can be an important consideration, especially if particles are moving slowly. Antialiasing points will cause the particles to move more smoothly as they cross pixel boundaries, since fragments with fractional alpha values will be generated. Another technique is to use the particle positions between two adjacent frames to orient a line centered at the particle's current position, and draw an antialiased line instead of a point. If the line's length and alpha are varied as a function of current velocity, you can create a motion blur effect.

If high quality is important and performance is not, or you have very good hardware support, the accumulation buffer can be used to generate excellent antialiasing and motion blur. The particles for a given frame can be rendered repeatedly and accumulated. The particle positions can be jittered for spatial antialiasing, and the particle re-rendered along its direction of motion can produce motion blur effects. For more information, see Section 9.5 in these notes, and the accumulation buffer paper in the 1990 SIGGRAPH Proceedings [43] reprinted in these course notes.

17.9.5 "Fat" Particles

Up until this point, we have dealt with very simple representations of particles. We do not have to limit ourselves to simple points, however. In OpenGL, points can be texture mapped and lit, providing ways to achieve more particle effects. It may also make sense to consider using small textured quads instead of points to represent particles for some systems. The quads can be textured with a texture map containing alpha values to describe its shape, transparency and color. Using more complex particles may allow you to use less particles to achieve the same visual effect, enhancing performance.

One problem with using quads or other surface primitives is that, unless you want to expose their planar nature, you will have to billboard them. Billboarding is rotating each quad so that it always faces the viewer. Since you control the orientation of the particles, this only becomes a problem when the viewing transformation changes. See Section 6.10 in these notes.

Some implementations have a billboarding extension, called `GL_sprite`, which will orient surfaces automatically. Implementation performance may vary, and since surfaces can all be oriented together, it may still be faster to billboard the surfaces yourself. Benchmark to be sure.

17.9.6 Particle Systems in a Scene

Particle systems can be difficult to integrate seamlessly into a complex scene. They are often not depth buffered, relying on the the accumulated light contributions of all the particles to create a particular effect. The rest of the scene will probably require depth buffering, however, so both the depth test and depth buffer update state needs to be managed within the scene. Although particles can be lit, it is extremely expensive to try to cause each particle to act as an OpenGL light source, especially since the number of simultaneous available OpenGL lights are limited. Instead a few light sources can be placed in the system to represent an overall lighting effect. Blending state must also be managed, since antialiased particles require alpha blending to work.

17.10 Precipitation

Precipitation effects such as rain and snow can be modeled and rendered using the particle techniques described above. The task can be broken down into several tasks:

1. Realistic particle rendering.
2. Computing particle dynamics.
3. Managing particle lifetime.

The basic particle rendering techniques are described in the preceding section. Using snowflakes as an example; individual flakes can be rendered as white colored points. Ideally the particle size should be rendered correctly under perspective projection as discussed for light points in Section 17.7. Since the real-life particles are subject to the effects of gravity, wind, thermal convection, etc, the modeled dynamics should include these effects. However, much of the complexity lies in the management of the particle lifetime. Again, considering the snow example, a running simulation must be maintained for the entire world, not just the portion that is currently visible. Particle dynamics may cause particles to move from a portion of the world which is not currently visible to the visible portion or vice versa. In the snow example, particles may shrink and disappear to mimic the melting effects of the sun.

One of the more difficult problems with managing the lifetime of particles is the end of life of the particle. Usually snowflakes accumulate to form a layer of snow over the objects upon which they fall. One way to model this is to terminate the particle dynamics when the particle strikes a surface (using a collision detection algorithm), but continue to draw it in its final position. A difficulty with this solution is that the number of particles which need to be drawn each frame will grow without bound. Another way to solve this problem is to draw the surfaces upon which the particles are falling as textured surfaces and when a particle strikes the surface, remove the particle from the dynamic system and incorporate it into the texture map used to render the surface. This solution allows the number of particles in the system to reach a steady state, but creates a new problem of efficiently managing the texture maps for the collision surfaces.

One way to maintain these texture maps is to use the rendering pipeline to update the maps. At the beginning of a simulation the texture map for a surface is clean. At the end of each frame, the particles which are to be retired this frame are drawn with an orthographic projection onto the textured surface (the viewpoint is perpendicular to the surface) using the current version of the texture and the resulting image replaces the current texture map. In order to avoid rendering artifacts when transitioning a particle from its live state to the texture map, it may be necessary to fade the live particle away over a few frames introducing a new limbo state for particles during this transition period.

Using a texture map for collided snow particles provides an efficient mechanism for maintaining a constant number of particles in the system and it works well for simulating the initial accumulation of precipitation on an uncovered surface. However, it does not serve as a realistic model for continued accumulation since it only simulates a one dimensional layer. To simulate continued accumulation, the model must be enhanced.

Changing our example from snow to rain, some of the properties of the precipitation change. Rain particles typically contain more mass than snow particles and are thus affected differently by gravity and wind. Heavy rain may be better simulated using short antialiased line segments rather than points to simulate motion blurring.

The initial accumulation of rain is a more complex problem than snow. In the case of snow, an opaque accumulation is built up over time. For rain, the rain drops are semi-transparent and they affect the surface characteristics and thus the surface shading of the collision surface in a more subtle manner. One way to model this effect is to create a texture map similar to the one created for the snow model. However, this map is used in conjunction with a multipass shading technique for the rest of the scene, partitioning the scene into two collections of pixels: those which are wet and those which are dry. The scene is drawn twice using two different shading models, one which renders objects which appear wet and another which renders objects with a dry appearance. The texture map is used to choose which computation to store in the framebuffer on a pixel by pixel basis.

Another method to reduce the rendering workload and increase the performance of the simulation is to reduce the number of particles using a “hollywood” technique. In this scheme rather than rendering particles throughout the entire volume a “curtain” of particles is rendered in front of the viewer. The use of motion blurring and fog along with lighting to simulate an overcast sky can make the illusion more convincing. It is still possible to simulate simple accumulation of precipitation by choosing points on collision surfaces at random (within the parameterization of the simulation) and blending them into texture maps as described above.

18 Tuning Your OpenGL Application

Tuning your software allows it to use hardware capabilities more effectively. Writing high-performance code is usually more complex than just following a set of rules. More often, it involves making trade-offs between special functionality, quality, and performance.

Since different hardware accelerators achieve optimal performance in different ways, not all rules apply in all cases. Some performance rules of thumb are applicable to most every OpenGL implementation – software or hardware – and others can be hardware-specific. This section provides many hints that may be used to tune your OpenGL application for optimal performance.

18.1 What Is Pipeline Tuning?

Traditional software tuning focuses on finding and tuning hot spots, the 10% of the code in which a program spends 90% of its time. Most graphics hardware accelerators are arranged in a pipeline, where one stage may perform vertex transformation and lighting while another draws the actual pixels into the framebuffer. Because these stages operate in parallel, it is appropriate to use a different approach: look for bottlenecks – overloaded stages that are holding up other processes.

At any time, one stage of the pipeline is the bottleneck. Reducing the time spent in that bottleneck is the best way to improve performance. Conversely, doing work that further narrows the bottleneck, or that creates a new bottleneck somewhere else, can further degrade performance.

If different parts of the hardware are responsible for different parts of the pipeline, the workload may instead be increased at one part of the pipeline without degrading performance, as long as that part does not become a new bottleneck. In this way, an application can sometimes be altered to draw, for example, a higher-quality image with no performance degradation.

Different programs (or portions of programs) stress different parts of the pipeline, so it is important to understand which elements in the graphics pipeline are the bottlenecks for your program.

Note that in a software implementation, the CPU does all the work. As a result, it does not make sense to increase the work for any stage if another is using more CPU time; you would be increasing the total amount of work for the CPU and decreasing performance.

18.1.1 Three-Stage Model of the Graphics Pipeline

The graphics pipeline consists of three conceptual stages. All three parts may be implemented in software or parts of the pipeline may be performed by a hardware graphics accelerator. The conceptual model is useful in either case: it helps you to know where your application spends its time. The stages are:

- The **application program** running on the CPU, feeding commands to the graphics subsystem (always on the CPU)
- The **geometry subsystem**, which performs per-vertex operations such as coordinate transformations, lighting, texture coordinate generation, and clipping (may be hardware-accelerated)
- The **raster subsystem**, which performs per-pixel operations such as the simple operation of writing color values into the framebuffer, or more complex operations like depth buffering, alpha blending, and texture mapping (may be hardware accelerated)

The amount of work required from the different pipeline stages varies depending on the application. For example, consider a program that draws a small number of large polygons. Because there are only a few polygons, the pipeline stage that performs geometry operations is lightly loaded. Because those few polygons cover many pixels on the screen, the pipeline stage that does rasterization is heavily loaded.

In this example, you must speed up the rasterization stage, either by drawing fewer pixels, or by drawing pixels in a way that takes less time by turning off modes like texturing, blending, or depth-buffering. In addition, because spare capacity is available in the per-polygon stage, you may be able to increase the workload at that stage without degrading performance. For example, use a more complex lighting model, or define geometries such that they remain the same size but look more detailed because they are composed of a larger number of polygons.

18.1.2 Finding Bottlenecks in Your Application

The basic strategy for isolating bottlenecks is to measure the time it takes to execute part or all of program and then change the code in ways that add or subtract work at a single point in the graphics pipeline. If changing the amount of work at a given stage does not alter performance appreciably, that stage is not the bottleneck. If there is a noticeable difference in performance, you have found a bottleneck.

Application Bottlenecks To see if your application is the bottleneck, remove as much graphics work as possible, while preserving the behavior of the application in terms of the number of instructions executed and the way memory is accessed. Often, changing just a few OpenGL calls is a sufficient test. For example, replacing the vertex and normal calls `glVertex3fv` and `glNormal3fv` with color subroutine calls (`glColor3fv`) preserves the CPU behavior while eliminating all drawing and lighting work in the graphics pipeline. If making these changes does not significantly improve performance, then your application is the bottleneck.

Geometry Bottlenecks Programs that create bottlenecks in the geometry (per-vertex) stage are termed *transform limited*. To test for bottlenecks in geometry operations, change the program so that the application code runs at the same speed and the same number of pixels are filled, but the geometry work is reduced. For example, if you are using lighting, call `glDisable` with a `GL_LIGHTING` argument to temporarily turn off lighting. If performance improves, your application has a geometry bottleneck. For more information, see "Tuning the Geometry Subsystem."

On some of the faster hardware accelerators the bus between the CPU and the graphics hardware can limit the number of polygons sent from the application to the geometry subsystems. If removing the `glColor3fv` or `glNormal3fv` calls shows a speed improvement on such a system, the bus may be the bottleneck.

Rasterization Bottlenecks Programs that cause bottlenecks at the rasterization (per-pixel) stage in the pipeline are *fill limited*. To test for bottlenecks in rasterization operations, shrink objects or make the window smaller to reduce the number of active pixels. This technique will not work if your program alters its behavior based on the sizes of objects or the size of the window. You can also reduce the work done per pixel by turning off per-pixel operations such as depth-buffering, texturing, or alpha blending. If any of these experiments speed up the program, it has a fill bottleneck. For more information, see "Tuning the Raster Subsystem."

Many programs draw a variety of things, each of which stress different parts of the system. Decompose such a program into pieces and time each piece. You can then focus on tuning the slowest pieces.

Since correct double buffering waits for the vertical retrace of the monitor before switching the buffer, you will only be able to time your application in units of the monitor refresh rate (e.g. 1/60 of a second), unless you run your tests in single-buffered mode. Single buffered behavior can be achieved with a double buffered visual by drawing to the *front* buffer. Screen clears and all the other normal operations can remain the same.

Table 8 provides an overview of factors that may limit rendering performance and the part of the pipeline to which they belong.

Performance Parameter	Pipeline Stage
Amount of data per polygon	All stages
Application overhead	Application
Transform rate and geometry mode setting	Geometry subsystem
Total number of polygons in a frame	Geometry and raster subsystem
Number of pixels filled	Raster subsystem
Fill rate for the current mode settings	Raster subsystem
Duration of screen and/or depth buffer clear	Raster subsystem

Table 8: Factors Influencing Performance

18.1.3 Measuring Depth Complexity

Finding depth complexity, or how many fragments were generated for each pixel in a rendered scene, is important for analyzing rasterization performance. It indicates how well polygons are distributed across the framebuffer and how many fragments were generated and discarded – clues for application tuning.

One way to show depth complexity is to use the color values of the pixels in the scene to indicate the number of times a pixel was written. It is relatively easy to draw an image representing depth complexity with the stencil buffer. The basic approach is simple. Increment a pixel's stencil value every time the pixel is written. When the scene is finished, read back the stencil buffer and display it in the color buffer, color coding the different stencil values.

This technique generates a count of the number of fragments generated for each pixel, whether the depth test failed or not. By changing the stencil operations, a similar technique could be used to count the number of fragments discarded after failing the depth test or to count the number of times a pixel was covered by fragments passing the depth test.

Here's the procedure in more detail:

1. Clear the depth and stencil buffer:

   ```
   glClear(GL_STENCIL_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   ```

2. Enable stenciling:

   ```
   glEnable(GL_STENCIL_TEST);
   ```

3. Set up the proper stencil parameters:

   ```
   glStencilFunc(GL_ALWAYS, 0, 0);
   glStencilOp(GL_KEEP, GL_INCR, GL_INCR);
   ```

4. Draw the scene.
5. Read back the stencil buffer with `glReadPixels` using `GL_STENCIL_INDEX` as the format argument.
6. Draw the stencil buffer to the screen using `glDrawPixels` with `GL_COLOR_INDEX` as the format argument.

You can control the mapping of stencil values to colors by `glPixelMap`. You can map the stencil values to either RGBA or color index values, depending on the type of color buffer to which you're writing. In color index mode, you must turn on the color mapping with `glPixelTransferi(GL_MAP_COLOR, GL_TRUE)`.

18.2 Optimizing Your Application Code

18.2.1 Optimize Cache and Memory Usage

On most systems, memory is structured in a hierarchy that contains a small amount of faster, more expensive memory at the top (e.g., CPU registers) and a large amount of slower memory at the base (e.g., hard disks). As memory is referenced, it is automatically copied into higher levels of the hierarchy, so data that is referenced most often migrates to the fastest memory locations.

The goal of machine designers and programmers is to maximize the chance of finding data as high up in this memory hierarchy as possible. To achieve this goal, algorithms for maintaining the hierarchy, embodied in the hardware and the operating system, assume that programs have locality of reference in both time and space; that is, programs are much more likely to access a location recently accessed or those nearby it, than elsewhere. Performance increases if you respect the degree of locality required by each level in the memory hierarchy.

Minimizing Cache Misses Most CPUs have first-level instruction and data caches on chip and many have second-level caches that are bigger but somewhat slower. Memory accesses are much faster if the data is already loaded into the first-level cache. When your program accesses data that is not in one of the caches, a *cache miss* occurs. This causes a block of consecutively addressed words, including the data that your program just accessed, to be loaded into the cache. Since cache misses are costly, you should try to minimize them, using these tips:

- Keep frequently accessed data together. Store and access frequently used data in flat, sequential data structures and avoid pointer indirection. This way, the most frequently accessed data remains in the first-level cache as much as possible.
- Access data sequentially. Each cache miss brings in a block of consecutively addressed words of needed data. If you are accessing data sequentially then each cache miss will bring in n words (where n is system dependent); if you are accessing only every nth word, then you will constantly be bringing in unneeded data, degrading performance.
- Avoid simultaneously traversing several large buffers of data, such as an array of vertex coordinates and an array of colors within a loop since there can be cache conflicts between the buffers. Instead, pack the contents into one buffer whenever possible. If you are using vertex arrays, try to use interleaved arrays. (For more information on vertex arrays see “Rendering Geometry Efficiently.”)

Some framebuffers have cache-like behaviors as well. It is a good idea to group geometry so that the drawing is done to one part of the screen at a time. Using triangle strips and polylines tends to do this while simultaneously offering other performance advantages as well.

18.2.2 Store Data in a Format That is Efficient for Rendering

Putting some extra effort into generating a simpler database makes a significant difference when traversing that data for display. A common tendency is to leave the data in a format that is good for loading or generating the object, but non-optimal for actually displaying it. For peak performance, do as much of the work as possible before rendering. This preprocessing is typically performed when an application can temporarily be non-interactive, such as at initialization time or when changing from a modeling to a fast-rendering mode.

See “Rendering Geometry Efficiently” and “Rendering Images Efficiently” for tips on how to store your geometric data and image data to make it more efficient for rendering.

Minimizing State Changes Your program will almost always benefit if you reduce the number of state changes. A good way to do this is to rearrange your scene data according to what state is set and render primitives with the same state settings together. Mode changes should be ordered so that the most expensive state changes occur least often. Typically it is expensive to change texture binding, material parameters, fog parameters, texture filter modes, and the lighting model. However, some experimentation will be required to determine which state settings are most expensive on your target systems. For example, on systems that accelerate rasterization, it may not be that expensive to change rasterization controls such as the depth test function and whether or not depth testing is enabled. However, if you are running on a system with software rasterization, this may cause cached graphics state, such as function pointers or automatically generated code, to be flushed and regenerated.

Your target OpenGL implementation may not optimize state changes that are redundant, so it is also important for your application to avoid setting the same state values twice, such as enabling lighting when it is already enabled.

18.2.3 Per-Platform Tuning

Many of the performance tuning techniques discussed here (e.g., minimizing the number of state changes and disabling features that are not required) are a good idea no matter what system you are targeting. Other tuning techniques are specific to particular system. OpenGL implementations vary widely, so inexpensive commands on one platform may be expensive on another. For example, before you sort your database based on state changes, you need to determine which state changes are the most expensive for each system on which you are interested in running.

In addition, you may want to modify the behavior of your program depending on which modes are fast. This is especially important for programs that must run faster than a particular frame rate. Features may need to be disabled in order to maintain interactivity. For example, if a particular texture mapping environment is slow on one of your target systems, you may need to disable texture mapping or change the texture environment whenever your program is running on that platform.

Before you can tune your program for each of the target platforms, you need to characterize those platforms' performance. This is not always straightforward. Often a particular device is able to accelerate certain features, but not all at the same time. Thus it is important to test the performance for combinations of features that you will be using. For example, a graphics adapter may accelerate texture mapping but only for certain texture parameters and texture environment settings. Even if all texture modes are accelerated, experimentation will be required to see how many textures you can use at once without causing the adapter to page textures in and out of the local memory.

An even more complicated situation arises if the graphics adapter has a shared pool of memory that is allocated to several tasks. For example, the adapter may not have a framebuffer deep enough to contain a depth buffer and a stencil buffer. In this case, the adapter would be able to accelerate both depth buffering and stenciling but not at the same time. Or perhaps, depth buffering and stenciling can both be accelerated but only for certain stencil buffer depths.

Typically, per-platform testing is done at initialization time. You should do some trial runs through your data with different combinations of state settings and calculate the time it takes to render in each case. You may want to save the results in a file so your program does not have to do this each time it starts up. You can find an example of how to measure the performance of particular OpenGL operations and save the results using the `isfast` program on the web site.

18.3 Tuning the Geometry Subsystem

18.3.1 Use Expensive Modes Efficiently

OpenGL offers many features that create sophisticated effects with excellent performance. However, these features have some performance cost, compared to drawing the same scene without them. Use these features only where their effects, performance, and quality are justified.

- Turn off features when they are not required. Once a feature has been turned on, it can slow the transform rate even when it has no visible effect.

  For example, the use of fog can slow the transform rate of polygons. When the polygons are too close to show fog, or when the fog density is set to zero, turn off fog explicitly with `glDisable(GL_FOG)`.

- Minimize mode changes. Be especially careful about expensive mode changes such as changing `glDepthRange` parameters and changing fog parameters when fog is enabled.

- For optimum performance of most software renderers and many hardware renderers as well, use flat shading. This reduces the number of lighting computations from one per-vertex to one per-primitive, and also reduces the amount of data that must be processed for each primitive. Keep in mind that long triangle strips approach one vertex per primitive and may show little benefit from flat shading.

18.3.2 Optimizing Transformations

OpenGL implementations are often able to optimize transform operations if the matrix type is known. Follow these guidelines to achieve optimal transform rates:

- Use `glLoadIdentity` to initialize a matrix, rather than loading your own copy of the identity matrix.

- Use specific matrix calls such as `glRotate`, `glTranslate`, and `glScale` rather than composing your own rotation, translation, or scale matrices and calling `glLoadMatrix` and/ or `glMultMatrix`.

18.3.3 Optimizing Lighting Performance

OpenGL offers a large selection of lighting features. The penalties some features carry may vary depending on the hardware you're running on. Be prepared to experiment with the lighting configuration.

As a general rule, use the simplest possible lighting model: a single infinite light with an infinite viewer. For some local effects, try replacing local lights with infinite lights and a local viewer. Keep in mind, however, that not all rules listed here increase performance for all architectures.

Use the following settings for peak performance lighting:

- Single infinite light.

- Nonlocal viewing. Set `GL_LIGHT_MODEL_LOCAL_VIEWER` to `GL_FALSE` in `glLightModel` (the default).

- Single-sided lighting. Set `GL_LIGHT_MODEL_TWO_SIDE` to `GL_FALSE` in `glLightModel` (the default).

- If two-sided lighting is used, use the same material properties for front and back by specifying `GL_FRONT_AND_BACK`.

- Do not use per-vertex color.

- Disable `GL_NORMALIZE`. Since it is usually only necessary to renormalize when the model-view matrix includes a scaling transformation, consider preprocessing the scene to eliminate scaling.

In addition, follow these guidelines to achieve peak lighting performance:

- Avoid using multiple lights.

  There may be a sharp drop in lighting performance when adding lights.

- Avoid using local lights.

  Local lights are noticeably more expensive than infinite lights.

- Use positional light sources rather than spot lights.

  If local lights must be used, a positional light is less expensive than a spot light.

- Do not change material parameters frequently.

  Changing material parameters can be expensive. If you need to change the material parameters many times per frame, consider rearranging the scene to minimize material changes. Also consider using `glColorMaterial` if you need to change some material parameters often, rather than using `glMaterial` to change parameters explicitly. Changing material parameters inside a `glBegin`/`glEnd` sequence can be more expensive than changing them outside.

  The following code fragment illustrates how to change ambient and diffuse material parameters at every polygon or at every vertex:

  ```
  glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
  glEnable(GL_COLOR_MATERIAL);
  /* Draw triangles: */
  glBegin(GL_TRIANGLES);
  /* Set ambient and diffuse material parameters: */
  glColor4f(red, green, blue, alpha);
  glVertex3fv(...);glVertex3fv(...);glVertex3fv(...);
  glColor4f(red, green, blue, alpha);
  glVertex3fv(...);glVertex3fv(...);glVertex3fv(...);
  ...
  glEnd();
  ```

- Avoid local viewer.

  Local viewing: Setting `GL_LIGHT_MODEL_LOCAL_VIEWER` to `GL_TRUE` with `glLightModel`, while using infinite lights only, reduces performance by a small amount. However, each additional local light noticeably degrades the transform rate.

- Disable two-sided lighting.

  Two-sided lighting illuminates both sides of a polygon. This is much faster than the alternative of drawing polygons twice. However, using two-sided lighting can be significantly slower than one-sided lighting for a single rendering of an object.

- Disable `GL_NORMALIZE`.

  If possible, provide unit-length normals and do not call `glScale` to avoid the overhead of `GL_NORMALIZE`. On some OpenGL implementations it may be faster to simply rescale the normal, instead of renormalizing it, when the modelview matrix contains a uniform scale matrix. The normal rescaling functionality in OpenGL 1.2, or the `EXT_rescale_normal` extension for older OpenGL versions, can be used to improve the performance of this case. If it is supported, you can enable `GL_RESCALE_NORMAL_EXT` and the normal will be rescaled making re- normalization unnecessary.

- Avoid changing the `GL_SHININESS` material parameter if possible.

  Some portions of the lighting calculation may be approximated with a table, and changing the `GL_SHININESS` value may force those tables to be regenerated.

18.3.4 Advanced Geometry-Limited Tuning Techniques

This section describes advanced techniques for tuning transform-limited drawing. Follow these guidelines to draw objects with complex surface characteristics:

- Use texture to replace complex geometry.

  Texture mapping can be used instead of extra polygons to add detail to a geometric object. This can greatly simplify geometry, resulting in a net speed increase and an improved picture, as long as it does not cause the program to become fill-limited. However, since many hardware implementations *are* slower to fill textured pixels than non-textured pixels, large areas to be covered with a simple texture can often be drawn faster if drawn as geometry.

- Use textured polygons as single-polygon billboards.

  Billboards are polygons that are fixed at a point and rotated about an axis, or about a point, so that the polygon always faces the viewer. Billboards can be used for distant objects to save geometry. Section 6.10 discusses how to render billboards.

- Use `glAlphaFunc` in conjunction with one or more textures to give the effect of rather complex geometry on a single polygon.

  Consider drawing an image of a complex object by texturing it onto a single polygon. Set alpha values to zero in the texture outside the image of the object. (The edges of the object can be antialiased by using alpha values between zero and one.) Orient the polygon to face the viewer. To prevent pixels with zero alpha values in the textured polygon from being drawn, call `glAlphaFunc(GL_NOTEQUAL, 0.0)`.

  This effect is often used to create objects like trees that have complex edges or many holes through which the background should be visible (or both).

- Eliminate objects or polygons that will be out of sight or too small to see. Section 5 discusses some techniques for occlusion culling.

18.4 Tuning the Raster Subsystem

An explosion of both data and operations is required to rasterize a polygon as individual pixels. Typically, the operations include depth comparison, Gouraud shading, color blending, logical operations, texture mapping, and possibly antialiasing. The following techniques can improve performance for a fill-limited applications.

18.4.1 Using Backface/Frontface Removal

To reduce fill-limited drawing, use backface and frontface removal. For example, if you are drawing a sphere, half of its polygons are backfacing at any given time. Backface and frontface removal is done after transformation calculations but before per-fragment operations. This means that backface removal may make transform-limited polygons somewhat slower, but make fill-limited polygons significantly faster. You can turn on backface removal when you are drawing an object with many backfacing polygons, then turn it off again when drawing is completed. Back face removal has the added advantage of eliminating x-fighting problems on objects with sharp edges.

18.4.2 Minimizing Per-Pixel Calculations

Another way to improve fill-limited drawing is to reduce the work required to render fragments.

Avoid Unnecessary Per-Fragment Operations Turn off per-fragment operations for objects that do not require them, and structure the drawing process to minimize their use without causing excessive toggling of modes. For example, if you are using alpha blending to draw some partially transparent objects, make sure that you disable blending when drawing the opaque objects. Also, if you enable alpha test to render textures with holes through which the background can be seen, be sure to disable alpha testing when rendering textures or objects with no holes. It also helps to sort primitives so that primitives that require alpha blending or alpha test to be enabled, are drawn at the same time (and hopefully *after* all non-transparent primitives).

Use Simple Fill Algorithms for Large Polygons If you are drawing very large polygons such as "backgrounds," your performance will be improved if you use simple fill algorithms. For example, you should set `glShadeModel` to `GL_FLAT` if smooth shading is not required. Also, disable per-fragment operations such as depth buffering, if possible. If you need to texture the background polygons, consider using `GL_REPLACE` for the texture environment. Keep in mind that on many architectures, a clear operation can be significantly faster than drawing large polygons.

Use the Depth Buffer Efficiently Any rendering operation can become fill-limited for large polygons. Clever structuring of drawing can eliminate or minimize per-pixel depth buffering operations. For example, if large backgrounds are drawn first, they do not need to be depth buffered. It is better to disable depth buffering for the backgrounds and then enable it for other objects where it is needed.

Games and flight simulators often use this technique. The sky and ground are drawn with depth buffering disabled, then the polygons lying flat on the ground (runway and grid) are drawn without suffering a performance penalty. Finally, depth buffering is enabled for drawing the mountains and airplanes.

There are many other special cases in which depth buffering might not be required. For example, terrain, ocean waves, and 3D function plots are often represented as height fields (X-Y grids with one height value at each lattice point). It's straightforward to draw height fields in back-to-front order by determining which edge of the field is furthest away from the viewer, then drawing strips of triangles or quadrilaterals parallel to that starting edge and working forward. The entire height field can be drawn without depth testing provided it does not intersect any piece of previously-drawn geometry. Depth values need not be written at all, unless subsequently-drawn depth buffered geometry might intersect the height field; in that case, depth values for the height field should be written, but the depth test can be avoided by calling `glDepthFunc(GL_ALWAYS)`.

18.4.3 Optimizing Texture Mapping

Follow these guidelines when rendering textured objects:

- Avoid frequent switching between texture maps. If you have many small textures, consider combining them into a single larger, mosaiced texture. Rather than switching to a new texture before drawing a textured polygon choose texture coordinates that select the appropriate small texture tile within the large texture.
- Use texture objects to encapsulate texture data. Place all the `glTexImage` calls (including mipmaps) required to completely specify a texture and the associated `glTexParameter` calls (which set texture properties) into a texture object and bind this texture object to the rendering context. This allows the implementation to compile the texture into a format that is optimal for rendering and, if the system accelerates texturing, to efficiently manage textures on the graphics adapter.
- Try to keep texture references localized between polygons. Some implementations use caching to optimize texture mapped rendering. Keeping the texture references localized when sending a batch of polygons to OpenGL can reduce the cache misses.
- If possible, use `glTexSubImage*D` to replace all or part of an existing texture image rather than the more costly operations of deleting and creating an entire new image.

- Call `glAreTexturesResident` to make sure that all your textures are resident during rendering. (On systems where texturing is done on the host, `glAreTexturesResident` always returns `GL_TRUE`.) If necessary, reduce the size or internal format resolution of your textures until they all fit into memory. If such a reduction creates intolerably fuzzy textured objects, you may use higher resolutions and specify which textures are important to keep in texture memory by using `glPrioritizeTextures`.
- Use smaller texel sizes. There is often a tradeoff between texel size and the speed of texture filtering, with smaller texel sizes typically performing better. Applications should try to minimize the width of a texel internal format to something like `GL_RGBA4` or `GL_RGB5_A1` for color textures and 8 bit components for luminance or luminance alpha textures unless the application requires the extra color resolution.
- Avoid expensive texture filter modes. On some systems, trilinear filtering is much more expensive than point sampling or bilinear filtering.

18.4.4 Clearing the Color and Depth Buffers Simultaneously

The most basic per-frame operations are clearing the color and depth buffers. On some systems, there are optimizations for common special cases of these operations.

Whenever you need to clear both the color and depth buffers, do not clear each buffer independently. Instead use `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`.

Also, be sure to disable dithering before clearing.

18.5 Rendering Geometry Efficiently

18.5.1 Using Peak-Performance Primitives

This section describes how to draw geometry with optimal primitives. Consider these guidelines to optimize drawing:

- Use connected primitives (line strips, triangle strips, triangle fans, and quad strips).

  Connected primitives are desirable because they reduce the amount of data both stored and transferred, and the amount of per-polygon or per-line work done by the OpenGL. Be sure to put as many vertices as possible in a `glBegin`/ `glEnd` sequence to amortize the cost of a `glBegin` and `glEnd`.
- Avoid using `glBegin(GL_POLYGON)`.

  When rendering independent triangles, use `glBegin(GL_TRIANGLES)` instead of `glBegin(GL_POLYGON)`. Also, when rendering independent quadrilaterals, use `glBegin(GL_QUADS)`.
- Batch primitives between `glBegin` and `glEnd`.

  Use a single call to `glBegin(GL_TRIANGLES)` to draw multiple independent triangles rather than calling `glBegin(GL_TRIANGLES)` multiple times. Also, use a single call to `glBegin(GL_QUADS)` to draw multiple independent quadrilaterals, and a single call to `glBegin(GL_LINES)` to draw multiple independent line segments.
- Use "well-behaved" polygons–convex and planar, with only three or four vertices.

  Concave and self-intersecting polygons must be tessellated by the GLU library before they can be drawn, and are therefore prohibitively expensive. Nonplanar polygons and polygons with large numbers of vertices are more likely to exhibit shading artifacts.

  If your database has polygons that are not well-behaved, perform an initial one-time pass over the database to transform the troublemakers into well- behaved polygons and use the new database for rendering. You can store the results in OpenGL display lists. Using connected primitives results in additional gains.

- Minimize the data sent per vertex.

  Polygon rates can be affected directly by the number of normals or colors sent per polygon. Setting a color or normal per vertex, regardless of the `glShadeModel` used, may be slower than setting only a color per polygon, because of the time spent sending the extra data and resetting the current color. The number of normals and colors per polygon also directly affects the size of a display list containing the object.

- Group like primitives and minimize state changes to reduce pipeline revalidation.

- Keep primitive data consistent.

  Try to send the same type of data for each vertex of a primitive. In other words, if the first vertex has an associated color or normal, the primitive can often be more efficiently processed if all the following vertices also have a color or normal.

- For wireframe objects, `GL_LINES`, `GL_LINE_STRIP` and `GL_LINE_LOOP` are likely to be significantly faster than drawing polygons as lines using `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`. First, the lines only are drawn once rather than twice. Second, lines representing the polygon edges of a closed object can easily be turned into long polylines which take up less space and are drawn more efficiently than individual lines.

18.5.2 Using Vertex Arrays

Vertex arrays are available in OpenGL 1.1. They offer the following benefits:

- The OpenGL implementation can take advantage of uniform data formats.

- The `glInterleavedArrays` call lets you specify packed vertex data easily. Packed vertex formats are typically faster for OpenGL to process.

- The `glDrawArrays` call reduces subroutine call overhead.

- The `glDrawElements` call reduces subroutine call overhead and also reduces per-vertex calculations because vertices may be reused. Be aware that using indexed vertices may introduce other problems with cache misses if the access pattern corresponding to the indexes is irregular enough. Indexed arrays are often most useful with implementations which perform the vertex processing on the CPU and may tend to degrade the performance of systems which have fast geometry processing in the accelerator if they become bottlenecked by the memory subsystem.

- Use the `EXT_compiled_vertex_array` extension if it is available. This extension allows you to lock down the portions of the arrays that you are using. This way the OpenGL implementation can DMA the arrays to the graphics adapter or reuse per-vertex calculations for vertices that are shared by adjacent primitives.

If you use `glBegin` and `glEnd` instead of `glDrawArrays` or `glDrawElements` calls, put as many vertices as possible between the `glBegin` and the `glEnd` calls.

18.5.3 Using Display Lists

You can often improve performance by storing frequently used commands in a display list. If you plan to redraw the same geometry multiple times, or if you have a set of state changes that need to be applied multiple times, consider using display lists. Display lists allow you to define the geometry and/or state changes once and execute them multiple times. Some graphics hardware may store display lists in dedicated memory or may store the data in an optimized form for rendering.

The biggest drawback of using display lists is data expansion. The display list contains an entire copy of all your data plus additional data for each command and for each list. As a result, tuning for display lists focuses mainly on reducing storage requirements. Performance improves if the data that is being traversed fits in the cache. Follow these rules to optimize display lists:

- Call `glDeleteLists` to delete display lists that are no longer needed. This frees storage space used by the deleted display lists and expedites the creation of new display lists.
- Avoid duplication of display lists. For example, if you have a scene with 100 spheres of different sizes and materials, generate one display list that is a unit sphere centered about the origin. Then reference the sphere many times, setting the appropriate material properties and transforms each time.
- Make the display lists as flat as possible, but be sure not to exceed the cache size. Avoid using an excessive hierarchy with many invocations to `glCallList`. Each `glCallList` invocation requires the OpenGL implementation to do some work (e.g., a table lookup) to find the designated display list. A flat display list requires less memory and yields simpler and faster traversal. It also improves cache coherency.

  On the other hand, excessive flattening increases the size. For example, if you're drawing a car with four wheels, having a hierarchy with four pointers from the body to one wheel is preferable to a flat structure with one body and four wheels.
- Avoid creating very small display lists. Very small lists may not perform well since there is some overhead when executing a list. Also, it is often inefficient to split primitive definitions across display lists.
- If appropriate, store state settings with geometry; it may improve performance.

  For example, suppose you want to apply a transformation to some geometric objects and then draw the result. If the geometric objects are to be transformed in the same way each time, it is better to store the matrix in the display list.

18.5.4 Balancing Polygon Size and Pixel Operations

The optimum size of polygons depends on the other operations going on in the pipeline:

- If the polygons are too large for the fill-rate to keep up with the rest of the pipeline, the application is fill-rate limited. Smaller polygons balance the pipeline and increase the polygon rate, allowing finer looking details and better lighting without changing the overall time to draw the object.
- If the polygons are too small for the rest of the pipeline to keep up with filling, then the application is transform limited. Larger and fewer polygons, or fewer vertices, balance the pipeline and increase the fill rate allowing the object to be drawn faster.

18.6 Rendering Images Efficiently

To improve performance when drawing pixel rectangles, follow these guidelines:

- Disable all per-fragment operations.
- Disable texturing and fog.
- Define images in the native hardware format so type conversion is not necessary.

- Know where the bottleneck is.

  Similar to polygon drawing, there can be a pixel-drawing bottleneck due to overload in host bandwidth, processing, or rasterizing. When all modes are off, the path is most likely limited by host bandwidth, and a wise choice of host pixel format and type pays off tremendously. For this reason, using type `GL_UNSIGNED_BYTE`, for the image components is sometimes faster.

  Zooming up pixels may create a raster bottleneck.

- A big pixel rectangle has a higher throughput (that is, pixels per second) than a small rectangle. Because the imaging pipeline is tuned to trade off a relatively large setup time with a high throughput, a large rectangle amortizes the setup cost over many pixels.

18.7 Tuning Animation

Tuning animation requires attention to some factors not relevant in other types of applications. This section discusses those factors.

18.7.1 Factors Contributing to Animation Speed

The smoothness of an animation depends on its frame rate. The more frames rendered per second, the smoother the motion appears.

Smooth animation also requires double buffering. In double buffering, one framebuffer holds the current frame, which is scanned out to the monitor by video hardware, while the rendering hardware is drawing into a second buffer that is not visible. When the new framebuffer is ready to be displayed, the system swaps the buffers. The system must wait until the next vertical retrace period between raster scans to swap the buffers, so that each raster scan displays an entire stable frame, rather than parts of two or more frames.

Frame rates must be integral multiples of the screen refresh time, which is 16.7 msec (milliseconds) for a 60-Hz monitor. If the draw time for a frame is slightly longer than the time for *n* raster scans, the system waits until the *n+1st* vertical retrace before swapping buffers and allowing drawing to continue, so the total frame time is (*n*+1)*16.7 msec. It may be very hard to make the final transition from one half of the display subsystem's refresh time to full speed, because you will need to speed up your program by a factor of at least two.

To summarize: A change in the time spent rendering a frame when double buffering has no visible effect unless it changes the total time to a different integer multiple of the screen refresh time.

If you want an observable performance increase, you must reduce the rendering time enough to take a smaller number of 16.7 msec raster scans. Alternatively, if performance is acceptable, you can add work without reducing performance, as long as the rendering time does not exceed the current multiple of the raster scan time.

To help monitor timing improvements, turn off double buffering by always drawing to the front buffer. If you do not, it is difficult to know if you are near a 16.7 msec boundary.

18.7.2 Optimizing Frame Rate Performance

The most important aid for optimizing frame rate performance is taking timing measurements in single-buffer mode only. For more detailed information, see "Taking Timing Measurements."

In addition, follow these guidelines to optimize frame rate performance:

- Reduce drawing time to a lower multiple of the screen refresh time.

  This is the only way to produce an observable performance increase.

- Perform non-graphics computation after swapping buffers.

  If an implementation allows control to return to a program while waiting to swap the color buffers, the program is free to do non-graphics computation. Therefore, the procedure for rendering a frame could be: call swapbuffers immediately after sending the last graphics call for the current frame, perform computation needed for the next frame, then execute OpenGL calls for the next frame.

- Do non-drawing work after a screen clear.

  Clearing a full screen can take time. If you make additional drawing calls immediately after a screen clear, you may fill up the graphics pipeline and force the program to stall. Instead, do some non-drawing work after the clear.

If you are rotating or otherwise moving an object at a fixed speed, it is wise to base the transformation on the amount of time spent rendering the frame rather than a fixed amount per frame, so that the motion does not speed up or slow down as scene complexity or viewing angle changes.

18.8 Taking Timing Measurements

Timing, or benchmarking, parts of your program is an important part of tuning. It helps you determine which changes to your code have a noticeable effect on the speed of your application.

To achieve performance that is demonstrably close to the best the hardware can achieve, you can first follow the more general tuning tips provided here, but you then need to apply a rigorous and systematic analysis.

18.8.1 Benchmarking Basics

A detailed analysis involves examining what your program is asking the system to do and then calculating how long that should take, based on the known performance characteristics of the hardware. Compare this calculation of expected performance with the performance actually observed and continue to apply the tuning techniques until the two match more closely. At this point, you have a detailed accounting of how your program spends its time, and you are in a strong position both to tune further and to make appropriate decisions considering the speed-versus-quality trade-off.

The following parameters determine the performance of most applications:

- Total number of polygons in a frame
- Transform rate for the given polygon type and mode settings
- Number of pixels filled
- Fill rate for the given mode settings
- Duration of color and depth buffer clear
- Duration of buffer swap
- Length of time spent in application overhead
- Number of attribute changes and time per change

18.8.2 Achieving Accurate Timing Measurements

Consider these guidelines to get accurate timing measurements:

- Take measurements on a quiet system. Verify that no unusual activity is taking place on your system while you take timing measurements. Terminate other applications. For example, do not have a clock or a network application like sendmail running while you are benchmarking.
- Choose timing trials that are not limited by the clock resolution.

  Use a high-resolution clock and make measurements over a period of time that's at least one hundred times the clock resolution. A good rule of thumb is to benchmark something that takes at least two seconds so that the uncertainty contributed by the clock reading is less than one percent of the total error. To measure something that's faster, write a loop to execute the test code repeatedly.
- Benchmark static frames.

  Verify that the code you are timing behaves identically for each frame of a given timing trial. If the scene changes, the current bottleneck in the graphics pipeline may change, making your timing measurements meaningless. For example, if you are benchmarking the drawing of a rotating airplane, choose a single frame and draw it repeatedly, instead of letting the airplane rotate, or make sure the rotation covers the same angles every time. Once a single frame has been analyzed and tuned, look at frames that stress the graphics pipeline in different ways, then analyze and tune them individually.
- Compare multiple trials.

  Run your program multiple times and try to understand variance in the trials. Variance may be due to other programs running, system activity, prior memory placement, or other factors.
- Call `glFinish` before reading the clock at the start and at the end of the time trial.

  This is important if you are using a machine with hardware acceleration because the graphics commands are put into a hardware queue in the graphics subsystem, to be processed as soon as the graphics pipeline is ready. The CPU can immediately do other work, including issuing more graphics commands until the queue fills up.

  When benchmarking a piece of graphics code, you must include in your measurements the time it takes to process all the work left in the queue after the last graphics call. Call `glFinish` at the end of your timing trial, just before sampling the clock. Also call `glFinish` before sampling the clock and starting the trial, to ensure no graphics calls remain in the graphics queue ahead of the process you are timing.

18.8.3 Achieving Accurate Benchmarking Results

To benchmark performance for a particular code fragment, follow these steps:

- Determine how many polygons are being drawn and estimate how many pixels they cover on the screen. Have your program count the polygons when you read in the database. To determine the number of pixels filled, start by making a visual estimate. Be sure to include surfaces that are hidden behind other surfaces, and notice whether or not backface elimination is enabled. For greater accuracy, use feedback mode and calculate the actual number of pixels filled or use the stencil buffer technique described in Section 18.1.3.
- Determine the transform and fill rates on the target system for the mode settings you are using. Refer to the product literature for the target system to determine some transform and fill rates. Determine others by writing and running small benchmarks.
- Divide the number of polygons drawn by the transform rate to get the time spent on per-polygon operations.

- Divide the number of pixels filled by the fill rate to get the time spent on per-pixel operations.
- Measure the time spent in the application. To determine time spent executing instructions in the application, stub out the OpenGL calls and benchmark your application.

This process takes some effort to complete. In practice, it is best to make a quick start by making some assumptions, then refine your understanding as you tune and experiment. Ultimately, you need to experiment with different rendering techniques and do repeated benchmarks, especially when the unexpected happens.

19 Portability Considerations

Think about portability from the beginning of the development cycle. Although this is a standard mantra for software development, it is important that OpenGL application developers in particular be aware of the flexibility of OpenGL and provide a way for their program to gracefully fall back onto an alternative algorithm or exit when a required implementation characteristic is not available.

19.1 General Concerns

Your OpenGL application should be at least a little flexible about the features it has available. A common goal is an application which can run well on almost all OpenGL platforms, and can also use the exceptional features on some platforms for high-speed and/or high-quality rendering.

It is unrealistic to expect an application developer to provide code that determines the best possible combination of modes and techniques for a given piece of hardware given both available features and those features' performance. However, a reasonable amount of time spent checking implementation characteristics at runtime can allow an application to better leverage an implementation with acceleration.

For example, one extreme is to develop an application that does not use the stencil buffer because the developer does not know if it will be available. The other extreme is to provide a fully general algorithm that uses 0, 1, or however many bits are available in the stencil buffer. A middle ground that maximizes portability, development time, and utilization of accelerated hardware might be to provide an algorithm that uses no stencil and an algorithm that uses 1 stencil bit and chooses between them at runtime based on querying the implementation.

19.1.1 Handle Runtime Feature Availability Carefully

OpenGL implementations vary widely in their support of buffer sizes and the availability of some buffers, such as stencil and the alpha channel, especially among PC hardware. Be prepared to proceed with a limited number of bits per component, and be prepared to drop back on an alternative algorithm if you need but cannot get, for example, the accumulation buffer *and* the stencil buffer.

Implementations may choose to provide some extensions but not others. Check at runtime for the extensions available to you and then choose whether the implementation has the capability for a more interesting algorithm, such as 3D texturing for volume rendering (Section 16.2). You can check for an extension by checking the result of `glGetString(GL_EXTENSIONS)` for the substring corresponding to the extension. Section 20 discusses using extensions in more detail.

When writing programs which automatically configure to the available extensions the program may use the dynamic linking capabilities of the underlying operating system to acquire addresses of the functions implementing the new commands. On most UNIX systems the `dlopen`, `dlsym`, and `dlclose` commands may be used to manipulate dynamic libraries and query functions. On Windows systems the commands `LoadLibrary`, `GetProcAddress`, and `FreeLibrary` provide similar functionality. Portable programs should use dynamic binding rather than relying on linking explicitly with extension function symbols.

Other capabilities to check include:

- The size available for textures, convolution kernels, color tables, and histograms.
- The precision of the accumulation buffer.
- The availability of specific resolutions of texture-internal formats.
- Whether hints are honored (`glHint`).
- The maximum recursion depth allowed during display list traversal.

- The maximum stack depth available for different OpenGL transforms.
- The maximum number of lights available.

Textures and other state elements that provide `PROXY` targets can test for the success of a state element binding without changing the actual values for that piece of state. You can identify the size available for one object by attempting to bind a very large object, then steadily reduce the size requested until the proxy parameters are accepted. A proxy binding that fails sets the state values for the proxy target to 0, while one that succeeds sets the proxy values to the parameters provided in the proxy call.

Note that the convolution extension does not provide a `PROXY` target but you can directly query the maximum width and height of the convolution kernel through `glGetConvolutionParameter*EXT` using `GL_MAX_CONVOLUTION_WIDTH_EXT` and `GL_MAX_CONVOLUTION_HEIGHT_EXT`.

19.1.2 Extensions and OpenGL Versioning

Some current OpenGL features were introduced first as extensions and eventually incorporated into the OpenGL core in a later version. For example, the `glPolygonOffset` command is both an extension and a part of OpenGL 1.1. Usually when an extension is incorporated into an OpenGL version, the extension suffixes from the commands and enumerants are removed and functionality is unchanged from the extension specification. In rare cases, the behavior diverges from the original extension when implementation experience suggests useful improvements. For example, the `EXT_polygon_offset`, `EXT_vertex_array` and `EXT_blend_logicop` extensions changed a little when they were added to OpenGL 1.1, whereas the `EXT_texture3D`, `EXT_texture_lod`, extensions remained essentially the same when their functionality was incorporated into OpenGL 1.2.

Some implementations of new versions of OpenGL may continue to support both the extension as well as the new version of the functionality. For cases where the core functionality behavior has diverged from the extension specification, the implemented extension behavior should still be compatible with the original extension specification.

While it is best to try to write applications to the latest version of OpenGL, sometimes it is desirable to support new and older versions of OpenGL as well as extensions within the same application in order to maximize the number of platforms the application will run on. To achieve this, the application must provide both compile-time and run-time guards to test for the existence of needed functionality for both the OpenGL version numbers and extension availability. At compile-time the OpenGL version can be tested with `#ifdef GL_VERSION_1_1` and `#ifdef GL_VERSION_1_2` and the run-time version can be tested with `glGetString(GL_VERSION)`. The first few characters of the version string will contain the current version number: 1.0, 1.1, or 1.2.

19.1.3 Source Compatibility Across OpenGL SDKs

Whether an implementation of OpenGL provides an extension or subset is determinable at runtime. However, the software development kit, including the link library and the headers, may not define some of the symbols or tokens used by an extension. If your application must be portable in source code form, it is important to place `#ifdef`/`#endif` guards around code that uses extensions.

For example, the preprocessor token `GL_EXT_texture3D` is defined in compile environments that export the 3D texture extension command and enumerants. Even if the implementation supports 3D texturing, you will not be able to compile or link your program if you use the symbols.

Keep this difference between compile-time and run-time availability in mind when designing both your source distribution and your application binary. Section 20 discusses this issue in more detail.

19.1.4 Characterize Platform Performance

Section 18 briefly discusses characterizing the performance of your application.

One of OpenGL's goals is to allow a program using the base API to "just work," no matter where it runs or is compiled. An implementation cannot be called `OpenGL` if it does not pass an exhaustive set of conformance tests that guarantee all the base features of OpenGL are available and are mathematically correct. However, that guarantee says nothing about the performance an application can expect. It will probably be necessary to check at run time some of the combinations of modes and states your application could use, and decide at that time which combination provides enough performance to be desirable.

Some typical features to check for performance availability include:

- Blending
- `GL_LINEAR` and `GL_*_MIPMAP_*` filters for texturing
- RGBA texture modes as opposed to color index textures
- Display lists if application data is largely static
- Vertex arrays and interleaved vertex arrays, if appropriate
- Convolution and other imaging extensions
- 3D textures

Example libraries `pdb` and `isfast` implement this notion of characterizing mode combinations. These libraries can be found by searching the OpenGL web site `www.opengl.org` and can be downloaded at the time of writing from `http://reality.sgi.com/gold/OpenGL/isfast.html`.

19.2 Windows and UNIX Portability

When writing samples and prototype code and even production applications, keep in mind that different UNIX implementations and Windows 95/NT have different APIs, provide different system services, and can even provide substantially different development environments (such as contents of `include` files, location of libraries, etc.). Here are a few things to look out for when writing a program under UNIX with the intent to port to Windows or other UNIX operating systems:

- The Win32 versions of the OpenGL and GLU header files depend on macros defined by including the `<windows.h>` header file. This forces you to do the following for Win32 portability:[10]

  ```
  #ifdef _WIN32
  #define WIN32_LEAN_AND_MEAN  /* somewhat limit Win32 pollution */
  #include <windows.h>
  #endif
  #include <GL/gl.h>
  #include <GL/glu.h>
  ```

  Unfortunately, including `<windows.h>` has the unfortunate side effect of introducing literally thousands of macros and type declarations into your compilation environment. This undesirable "name space pollution" can sometimes affect source code portability by conflicting with your program's own macros and types. This can particularly be a problem for UNIX programmers that are not familiar with all the junk that comes with including `<windows.h>`.

  One alternative is to include the `<GL/glut.h>` header. The GLUT header automatically includes `<GL/gl.h>` and `<GL/glu.h>` and guarantees to include these headers in a way that avoids introducing

[10] According to Microsoft's documentation, the pre-defined `_WIN32` macro is "Defined for applications for Win32. Always defined." This macro is therefore the most automatic way to conditionally compile code just for Win32 programs.

the name space pollution of including `<windows.h>`. If you use GLUT, your programs will automatically be more portable by simply including `<GL/glut.h>` and *not including* `<GL/gl.h>` or `<GL/glu.h>` directly (simply letting `<GL/glut.h>` include them).

- Avoid the identifiers `near` and `far`, which are reserved words in most Intel and Windows compilers. Common replacements are `nnear` and `ffar`.
- The math constant `M_PI` and related constants such as `M_PI_2` are not provided by at least one Win32 development environment. You may find adding the following code after `#include <math.h>` to be helpful:

```
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif
```

- Do not `#include <unistd.h>` as it contains UNIX-specific definitions. At the very least, check with your Windows environment before using functions or constants from `<unistd.h>`.
- The ANSI C library defined constants `EXIT_SUCCESS` and `EXIT_FAILURE` may not be available. You could include code to define these constants similar to the above code for `M_PI`.
- Single-precision versions of trigonometric functions such as `sinf` and `cosf` while desirable for performance may not be available on all platforms.
- If you need to generate random numbers, use the ANSI C function `rand`. Do not use the traditional UNIX functions `random` or `drand48` since they are not supported by Win32.
- When opening binary files such as an image file with `fopen`, be sure to use a mode string of `"wb"` for writing or `"rb"` for reading. Without the `b` option (it stands for binary), Win32 opens the file for access as a text file and performs translations of formfeed and linefeed characters. Using the `b` option suppresses these translations.
- If you use the GLU tessellator, use the `CALLBACK` calling convention identifier (this is really just a macro for the calling convention keyword `__cdecl` in Win32). You must do this for the callbacks to work correctly under Win32. For the benefit of UNIX environments that do not define a `CALLBACK` macro (because they do not need it!), after including `<GL/glu.h>`, include the following:

```
#ifndef CALLBACK
#define CALLBACK
#endif
```

Then using a GLU tessellator begin callback as an example say:

```
static void CALLBACK
begin(GLenum type, void *polyData)
{
     glBegin(type);
}
```

When registering the callback, say:

```
gluTessCallback(tess, GLU_TESS_BEGIN_DATA,
                (void (CALLBACK*)()) &begin);
```

This advice also applies to the other GLU routines that require callback functions to be supplied. These other routines are `gluQuadricCallback` and `gluNurbsCallback`.

- It is sometimes tempting in graphics programs to name a variable `quad`, short for quadrilateral. Avoid the temptation. Some operating systems such as IBM's AIX variant of UNIX define `quad` to be 64-bit data type in `<sys/types.h>` which is often implicitly included by many system header files.

A more in-depth list of portability considerations is available in the file `Portability.txt` in the GLUT source code distribution. GLUT is described in more detail in Appendix B.

19.3 3D Texture Portability

With the release of OpenGL 1.2, 3D texturing is now core feature of OpenGL, but the functionality is also available via the `EXT_texture3D` extension. Even when 3D texture maps are supported, the application writer must be careful to consider the level of support present in the application. Texture map size may be limited, and 3D mipmapping is sometimes not supported in hardware. Available internal and external formats and types may be restricted. All of these restrictions can be queried at run time, and with care, portable code can be produced.

Consider writing your 3D texture applications so that they revert to a 2D texturing mode if 3D textures are not supported. See Section 16.2 for an example of a 3D texture algorithm that will work, with lower quality, using 2D textures.

20 Using OpenGL Extensions

OpenGL is an *extensible* API, and these notes make references to OpenGL extensions that enhance OpenGL's base functionality.

By design, OpenGL implementors are free to extend OpenGL's basic rendering functionality with new rendering operations. This extensibility was one of OpenGL's original design goals. Scores of OpenGL extensions have been specified and implemented. These extensions provide OpenGL application developers with new rendering features above and beyond the features specified in the official OpenGL standard. OpenGL extensions keep the OpenGL API current with the latest innovations in graphics hardware and rendering algorithms.

This section describes the OpenGL extension mechanism. You will learn not just how extensions are used and documented but also how to use extensions portably in your programs. Particular attention is paid to using OpenGL extensions in the Win32 environment because of the additional hoops that Win32 makes you jump through.

20.1 How OpenGL Extensions are Documented

An OpenGL extension is defined by its specification. These specifications are typically written as standard ASCII text files. OpenGL extension specifications are written by and for OpenGL implementors. A well-written OpenGL specification is documented to the level of detail needed for a hardware designer and/or OpenGL library engineer to implement unambiguously the extension. This means that OpenGL application programmers should not expect an extension's specification to justify fully why the functionality exists or explain how an OpenGL application would go about using the functionality. An OpenGL extension specification is not a tutorial on how to use the particular extension. Still, being able to read and understand an OpenGL extension specification helps you, the application programmer, fully understand an OpenGL extension's functionality.

20.2 Finding OpenGL extension specifications

The latest public OpenGL specifications can be found on the `www.opengl.org` web site. Note that extension specifications are updated from time to time based on reviews and implementation feedback. In the case of certain proprietary OpenGL extensions, it may be necessary to contact the OpenGL vendor that developed the extension for the extension's specification.

20.3 How to Read an OpenGL Extension Specification

When reading an OpenGL extension specification, it helps to be familiar with the original OpenGL specification. The operation of an OpenGL extension is described as additions and changes to the core OpenGL specification. Having a copy of the core OpenGL specification handy is a good idea when reviewing an OpenGL specification.

OpenGL extension specifications consist of multiple sections. There is a common form established by convention used by nearly all OpenGL extension specifications. Often within a specification, the `gl` and `GL` prefixes on routine names and tokens are assumed. The following describes the purpose of the most common sections in the order that they normally appear in extension specifications:

Name Lists the official name of the extension. This name uses underscores instead of spaces between words. The name also begins with a prefix that indicates who developed the extension. This prefix helps to avoid naming conflicts if two independent groups implement a similar extension. It also helps identity who is promoting use of the extension. For example: `SGIS_point_parameters` was an extension proposed by Silicon Graphics. The `SGIS` prefix belongs to Silicon Graphics. SGI uses the `SGIS` prefix to indicate that the extension is specialized and may not be available on all SGI hardware. Other prefixes in use are:

ARB - Extensions officially approved by the OpenGL Architectural Review Board

EXT - Extensions agreed upon by multiple OpenGL vendors

ES - Evans and Sutherland

HP - Hewlett-Packard

IBM - International Business Machines

INTEL - Intel

KTX - Kinetix (maker of 3D Studio Max)

MESA - Brian Paul's freeware portable OpenGL implementation

NV - NVIDIA Corporation

SGI - Silicon Graphics

SGIS - Silicon Graphics (limited set of machines)

SGIX - Silicon Graphics (experimental)

SUN - Sun Microsystems

WIN - Microsoft

Note that the `SGIS_point_parameters` extension has since been standardized by other OpenGL vendors. So now there is also an `EXT_point_parameters` extension with the same basic functionality as the `SGIS` version. The `EXT` prefix indicates that multiple vendors have agreed to support the extension. Successful OpenGL extensions are often promoted to `EXT` or `ARB` extensions or made an official part of OpenGL in a future revision to the core OpenGL specification. Almost all of the new functionality in OpenGL 1.1 and 1.2 showed up first as OpenGL extensions.

Name Strings The name string or strings is used to indicate that the extension is supported by a given OpenGL implementation. Applications can query the `GL_EXTENSIONS` string with OpenGL's `glGetString` to determine what extensions are available. OpenGL also supports the idea of window system dependent extensions. Core OpenGL extension name strings are generally prefixed with `GL_` while window system dependent extensions are prefixed with `GLX_` for the X Window System or `WGL_` for Win32 based on what window system to which the extension applies. Note that there may be multiple strings if the extension provides both core OpenGL rendering functionality and window system dependent functionality.

In the case of the X Window System, support for GLX extensions is indicated by listing the GLX extension name in the string returned by `glxQueryExtensionsString`. Querying the core OpenGL extension string that requires an OpenGL rendering context be created and made current (calling `glGetString` assumes a current OpenGL context). However, using `glxQueryExtensionString` only requires a connection to an X server. Because the X Window System is client/server based, the OpenGL client library may support different extensions than the OpenGL server. For this reason, it is also possible to query the extensions supported by the client or server individually using `glxQueryClientString` and `glxQueryServerString` respectively. To actually use most GLX extensions, a GLX extension must be supported by both the OpenGL client and server (it is possible for an extension to be a pure client-side extension though). For this reason, the strings returned by `glxQueryClientString` and `glxQueryServerString` are intended for informational use only. The string returned by `glxQueryExtensionsString` is typically intersection of the extensions supported by both the client and server. This is the string you should check before you use a GLX extension.

WGL extensions are advertised through OpenGL's core extension string, the one returned by `glGetString`.

Version A source code control revision string to keep track of what version of the specification the given text file represents. It is important to make sure that you have the latest version of the extension specification in case there are any important changes. Normally the version string has the date the extension was last updated.

Number Each OpenGL extension is assigned a unique number. Silicon Graphics allocates these numbers to ensure that OpenGL extensions do not overlap in their usage of enumerants or protocol tokens. This number is only important to extension implementors.

Dependencies Often an extension specification builds on the functionality of pre-existing extensions. This section documents other extensions upon which the specified extension depends. Dependencies indicate that another extension "is required" to support the specified extension or that the specified extension "affects" the specification of another extension. When an extension affects the specification of another extension, the affecting extension is responsible for fully documenting the interactions between the two extensions.

The dependencies section often also indicates which version of the OpenGL core standard that the extension specification is based on. Later sections specify the extension based on updates to the relevant section of the particular OpenGL specification that the extension is based on.

You can often tell how important a given extension is to the evolution of OpenGL based on how many other extensions are listed that depend on or are affected by the given extension.

Overview The section provides a description, often terse and without justification, for the extension's specified functionality. If you are trying to figure out what the extension does, this is the most useful section of an OpenGL extension specification. Do not expect a tutorial though.

Issues Often there are issues that need to be resolved in the specification of an extension. This section documents open issues and states the resolution to resolved issues. These issues are often things of interest to the extension implementor, but can also help a programmer understand how the extension really works.

New Procedures and Functions This section lists the function prototypes for any new procedures and functions that the extension adds. Keep in mind that specifications often leave out the `gl` prefix when discussing routines. Also note that the extension's new functions will be suffixed with the same letters used as the prefix for the extension name.

New Tokens This section lists the tokens (also called enumerants) that the extension adds. The routines that accept each set of new enumerants are documented. The integer value of the enumerants is documented here. These values should be added to `<GL/gl.h>`. Keep in mind that specifications often leave out the `GL_` prefix when discussing enumerants. Also note that the extension's new enumerants will be suffixed with the same letters used as the prefix for the extension name.

Additions to Chapter *XX* of the 1.*X* Specification (*XXX*) These sections document how the core OpenGL specification should be amended to add the extension's functionality to the core OpenGL functionality. Notice that the exact version of the core OpenGL specification (such as 1.0, 1.1, or 1.2) is documented. The chapters typically amended by an extension specification are:

- Chapter 2 - OpenGL Operations
- Chapter 3 - Rasterization
- Chapter 4 - Fragments and the Framebuffer
- Chapter 5 - Special Functions
- Chapter 6 - State and State Requests

These sections are quite legalistic. They indicate precisely how the OpenGL specification wording should be amended or changed. Often tables within the specification are amended as well.

Additions to the GLX Specification If an extension has any window system dependent functionality affecting the GLX interface to the X Window System, these issues would be documented here.

GLX Protocol When implementing the extension for the X Window System, if any special X11 extension protocol for the GLX extension is required to support the extension, the protocol would be documented in this section. This section is only interesting to GLX protocol implementors because the GLX protocol is hidden from application programmers beneath the OpenGL API.

Dependencies on *XXX* These sections describe how the extension depends on some other extension that was listed in the *Dependencies* section. Usually the wording says that if the other extension is not supported, simply ignore the portion of this extension dealing with the dependent extension's state and functionality.

Errors If the extension introduces any new error conditions particular to the extension, they are documented here.

New State Extensions typically add new state variables to OpenGL's state machine. These new variables are documented in this section. The variable's get enumerant, type, get command, initial value, description, section of the specification describing the state variable's function, and the attribute group that the state belongs to are all documented in tables in this section.

New Implementation Dependent State Extensions may add implementation dependent state. These are typically maximum and minimum supported ranges for the extension functionality. For example, what is the widest line size supported by the extension. These values can be queried through OpenGL's `glGet` family of routines.

Backward Compatibility If the extension supersedes an older extension, issues surrounding backward compatibility with the older extension are documented in this section.

Note that these sections are merely established by convention. While the conventions for OpenGL extension specifications are normally followed, extensions vary in how closely they stick to the conventions. Generally, the more preliminary an extension is, the more loosely specified it is. Hopefully after sufficient review and even implementation, the specification language and format is improved to provide an unambiguous final specification.

20.4 Portably Using OpenGL Extensions

The advantage of using OpenGL extensions is getting access to cutting edge rendering functionality so you application can achieve higher performance and higher quality rendering. OpenGL extensions give you access to the latest features of the hottest new graphics hardware. The problem with OpenGL extensions is that lots of OpenGL implementations, particularly older implementations, will not support the extensions that you would like to use. When you write an OpenGL application that uses extensions, you should make sure that your application still works when the extension is not supported. At the very least your program should report that it requires whatever extension is missing and exit without crashing.

The first step to using OpenGL extensions is to locate the copy of the `<GL/gl.h>` header file that advertises the API interfaces for the extensions that you plan to use. Typically you can get this from your OpenGL implementation vendor or OpenGL driver vendor. You could also get the API interface prototypes and macros directly from the extension specifications, but getting the right `<GL/gl.h>` from your OpenGL vendor is definitely the preferred way.

You will notice that `<GL/gl.h>` sets C preprocessor macros to indicate whether the header advertises the interface of a particular extension or not. For example, the basic `<GL/gl.h>` supplied with Microsoft Visual C++ 4.2 has a section reading:

```
/* Extensions */
#define GL_EXT_vertex_array              1
#define GL_WIN_swap_hint                 1
#define GL_EXT_bgra                      1
#define GL_EXT_paletted_texture          1
#define GL_EXT_clip_disable              1
```

These macros indicate that the header file advertises the above five extensions. The `EXT_bgra` extension lets you read and draw pixels in the Blue, Green, Red, Alpha component order as opposed to OpenGL's standard RGBA color component ordering.[11] If you wanted to write a program to use the `EXT_bgra` extension, you could test that the extension is supported at compile time like this:

```
#ifdef GL_EXT_bgra
   glDrawPixels(width, height, GL_BGRA_EXT, GL_UNSIGNED_BYTE, pixels);
#endif
```

When `GL_EXT_bgra` is defined, you can expect to find the `GL_BGRA_EXT` enumerant defined. Note that if the `EXT_bgra` extension is not supported, expect the `glDrawPixels` line above to generate a compiler error because the standard *unextended* OpenGL header does not define the `GL_BGRA_EXT` enumerant.

So based on the extension name `#define` in `<GL/gl.h>`, you can write your code so that it can compile in the extension functionality if your development environment supports the extension's interfaces. The next problem is that even though your development environment may support the extension's interface at compile-time, at run-time, the target system where you run your application may not support the extension. In UNIX environments, different systems with different graphics hardware often support different sets of extensions. Likewise, in the Win32 environment, different OpenGL accelerated graphics boards will support different OpenGL extensions because they have different OpenGL drivers. The point is that you can not just assume a given extension is supported. You must make a run-time check to verify that the extension you wish to use is supported.

Assuming that your application thread is made current to an OpenGL rendering context, the following routine can be used to determine at run-time if the OpenGL implementation really supports a particular extension:

```
#include <GL/gl.h>
#include <string.h>

int
isExtensionSupported(const char *extension)
{
  const GLubyte *extensions = NULL;
  const GLubyte *start;
  GLubyte *where, *terminator;

  /* Extension names should not have spaces. */
  where = (GLubyte *) strchr(extension, ' ');
  if (where || *extension == '\0')
    return 0;

  extensions = glGetString(GL_EXTENSIONS);

  /* It takes a bit of care to be fool-proof about parsing the
     OpenGL extensions string.  Don't be fooled by sub-strings,
     etc. */
  start = extensions;
  for (;;) {
    where = (GLubyte *) strstr((const char *) start, extension);
    if (!where)
      break;
    terminator = where + strlen(extension);
    if (where == start || *(where - 1) == ' ')
      if (*terminator == ' ' || *terminator == '\0')
        return 1;
    start = terminator;
  }
```

[11] The functionality of the `EXT_bgra` extension is now an official part of OpenGL 1.2. The BGRA color component ordering is important because it matches the color component ordering of Win32's GDI 2D API and therefore many PC-based file formats use it.

```
  return 0;
}
```

With the isExtensionSupported routine, you can check if the current OpenGL rendering context supports a given OpenGL extension. To make sure that the EXT_bgra extension is supported before using it, you can do the following:

```
  /* At context initialization. */
  int hasBGRA = isExtensionSupported("GL_EXT_bgra");

  /* When trying to use EXT_bgra extension. */
#ifdef GL_EXT_bgra
  if (hasBGRA) {
    glDrawPixels(width, height, GL_BGRA_EXT, GL_UNSIGNED_BYTE, pixels);
  } else
#endif
  {
    /* No EXT_bgra so bail (or implement software workaround). */
    fprintf(stderr, "Needs EXT_bgra extension!\n");
    exit(1);
  }
```

Notice that if the EXT_bgra extension is lacking at either run-time or compile-time, the code above will detect the lack of EXT_bgra support. Sure the code is a bit messy, but the code above works. You can skip the compile-time check if you know what development environment you are using and you do not expect to ever compile with a <GL/gl.h> that does not support the extensions that your application uses. But the run-time check really should be performed since who knows on what system your program may end up getting run on.

20.5 Win32's Scheme for Getting Extension Function Pointers

Most OpenGL implementations support extension commands just like core commands. Assuming your OpenGL header file provides the function prototypes and enumerants for the extension you want to use, you simply compile your executable assuming the extension routines exist. The assumption then is that before calling any extension routines, your program will first check the GL_EXTENSIONS string value to verify that the OpenGL extension is supported. If the extension is supported, you can then safely call the extension's routines and use its enumerants. If not supported, your program must avoid using the extension.

In the case of using an extension's new routines, this works because most operating systems today support flexible shared libraries. A shared library delays the binding of a routine name to its executable function until the routine is first called when your program runs. This is known as a *run-time* link instead of a *compile-time* link. A problem occurs if you call an OpenGL extension routine that is not supported by your OpenGL run-time library. The result is a run-time link error that is generally a fatal program error. This is why it is so important to check the GL_EXTENSIONS string before using any extension. If you have first verified the extension is supported, then your program can safely call the extension's routines in full expectation that the system's run-time linker will invoke the extension routine correctly.

Unfortunately, the world's most popular operating system does not work so nicely. Win32 operating systems require special care to invoke OpenGL extension routines. Win32 programs using OpenGL link with a Microsoft-supplied shared library called OPENGL32.DLL. Because the library is supplied by Microsoft, OpenGL hardware vendors are not at liberty to change it to add extensions. Instead, OpenGL hardware vendors implement another hidden shared library known as an ICD or Installable Client Driver. When an ICD is installed the OPENGL32.DLL library passes most OpenGL calls directly to the driver. Unfortunately, because extension routines are not present in the OPENGL32.DLL shared library that OpenGL programs actually link with, is no way for the Win32 run-time linker to call the driver's extension routine automatically. The bottom line is that this makes using extensions more difficult in the Win32 environment.

The EXT_bgra example above showing how to safely detect and use the extension at run-time and compile-time is straightforward because the EXT_bgra simply adds two new enumerants (GL_BGRA_EXT and GL_BGR_EXT) and does not require any new routines.

Using an extension that includes new function call entry-points is harder in Win32 because you must first explicitly request the function pointer from the OpenGL ICD before you can call the OpenGL function.

The EXT_point_parameters extension provides eye-distance attenuation of OpenGL's point primitive. Section 17.9.2 discusses the extension as a means to render particle systems. Indeed, the extension is used by Id Software in Quake 2 for rendering particle systems. With the extension, firing weapon and explosions are rendered as huge clusters of OpenGL point primitives with OpenGL automatically adjusting the point size based on the distance of the particles from the viewer. Closer particles appear bigger; particles in the distance appear smaller. A particle whose size would be smaller than a pixel is automatically faded based on its sub-pixel size. Anyone that wants to see the improvement this extension brings to a 3D game should play Quake 2 on a PC with an OpenGL driver supporting the EXT_point_parameters extension. Start a gun battle and check out the particles!

The EXT_point_parameters extension adds two new OpenGL entry points called glPointParameterfEXT and glPointParameterfvEXT. These routines allow the application to specify the attenuation equation parameters and fade threshold. As explained, because of the way Microsoft chose to support OpenGL extension functions, an OpenGL application cannot simply link with these functions. The application must first use the wglGetProcAddress routine to query the function address and then call through the returned address to call the extension function.

First, declare function prototype typedefs that match the extension's entry points. For example:

```
#ifdef _WIN32
typedef void (APIENTRY * PFNGLPOINTPARAMETERFEXTPROC)
               (GLenum pname, GLfloat param);
typedef void (APIENTRY * PFNGLPOINTPARAMETERFVEXTPROC)
               (GLenum pname, const GLfloat *params);
#endif
```

Your <GL/gl.h> header file may already have these typedefs declared if your <GL/gl.h> defines the GL_EXT_point_parameters macro. Now declare global variables of the type of these function prototype typedefs like this:

```
#ifdef _WIN32
PFNGLPOINTPARAMETERFEXTPROC glPointParameterfEXT;
PFNGLPOINTPARAMETERFVEXTPROC glPointParameterfvEXT;
#endif
```

The names above exactly match the extension's function names. Once we use wglGetProcAddress to assign these function variables the address of the OpenGL driver's extension functions, we can call glPointParameterfEXT and glPointParameterfvEXT as if they were normal functions. You pass wglGetProcAddress the name of the routine as an ASCII string. Verify that the extension is supported and, if so, initialize the function variables like this:

```
  int hasPointParams = isExtensionSupported("GL_EXT_point_parameters");
#ifdef _WIN32
  if (hasPointParams) {
    glPointParameterfEXT = (PFNGLPOINTPARAMETERFEXTPROC)
      wglGetProcAddress("glPointParameterfEXT");
    glPointParameterfvEXT = (PFNGLPOINTPARAMETERFVEXTPROC)
```

```
      wglGetProcAddress("glPointParameterfvEXT");
  }
#endif
```

Note that before the code above is called, you should have a current OpenGL rendering context.

With the function variables properly initialized to the extension entry-points, you can use the extension like this:

```
  if (hasPointParams) {
    static GLfloat quadratic[3] = { 0.25, 0.0, 1/60.0 };
    glPointParameterfvEXT(GL_DISTANCE_ATTENUATION_EXT, quadratic);
    glPointParameterfEXT(GL_POINT_FADE_THRESHOLD_SIZE_EXT, 1.0);
  }
```

Be careful because the function returned by `wglGetProcAddress` is only guaranteed to work for the pixel format type of the OpenGL rendering context that was current when `wglGetProcAddress` was called. If you have multiple contexts created for different pixel formats, then keeping a single function addresses in a global variable as shown above may create problems. You may need to maintain distinct function addresses on a per-pixel format basis. Specifically, the Microsoft documentation for `wglGetProcAddress` warns:

> The [Microsoft] OpenGL library supports multiple implementations of its functions. Extension functions supported in one rendering context are not necessarily available in a separate rendering context. Thus, for a given rendering context in an application, use the function addresses returned by the `wglGetProcAddress` function only.
>
> The spelling and the case of the extension function pointed to by string must be identical to that of a function supported and implemented by OpenGL. Because extension functions are not exported by OpenGL, you must use `wglGetProcAddress` to get the addresses of vendor-specific extension functions.
>
> The extension function addresses are unique for each pixel format. All rendering contexts of a given pixel format share the same extension function addresses.

Win32's requirement that you use `wglGetProcAddress` is a real drag, but if you do everything right, using OpenGL extensions works and gives you access to amazing new OpenGL features. And let's be honest; `wglGet-ProcAddress` is hardly the only annoying and awkward thing about programming with the Win32 API. Still, by using the C preprocessor and coding carefully, you actually *can* write OpenGL programs that use OpenGL extensions and compile from the same source code for both UNIX and Win32 environments.

A List of Demo Programs

This list shows the demonstration programs available on the Programming with OpenGL: Advanced Rendering web site at:

`http://www.sgi.com/software/opengl/courses.html`

The programs are grouped by the sections in which they're discussed. Each line gives a short description of the program.

Modeling

- tvertex.c - show problems caused by t-vertices
- quad_decomp.c - shows example of quadrilateral decomposition
- tess.c - shows examples of sphere tessellation
- cap.c - shows how to cap the region exposed by a clipping plane
- csg.c - shows how to render CSG solids with the stencil buffer
- gen_normals.c - shows how to generate correct normals

Geometry and Transformations

- depth.c - compare screen and eye space z
- decal.c - shows how to decal coplanar polygons with the stencil buffer
- hiddenline.c - shows how to render wireframe objects with hidden lines
- stereo.c - shows how to generate stereo image pairs
- tile.c - shows how to tile images
- raster.c - shows how to move the current raster position off-screen
- frustum_z.c - shows an object and its place in view frustum
- inaccuracies.c - provides examples of precision inaccuracy problems
- hidden.c - shows how polygon offset works with depth range
- stereoview.c - shows how to do stereo viewing right
- clipwide.c - shows how to avoid clipping wide lines and points
- distort.c - shows how to correct projection distortion using texture
- locate.c - shows how to pick objects and highlight them

Occlusion Culling

- occull.c - shows how to compute an occlusion map and test against it

Texture Mapping

- mipmap_lines.c - shows different mipmap generation filters

- genmipmap.c - shows how to use the OpenGL pipeline to generate mipmaps
- textile.c - shows how to tile textures
- texpage.c - shows how to page textures
- mippage.c - shows how to page a mipmapped texture
- textrim.c - shows how to trim textures
- textext.c - shows how draw characters with texture maps
- terrain.c - shows how to do elevation color coding and metrics
- contour.c - shows hot to do contouring
- projtex.c - shows how to use projective textures
- cyl_billboard.c - shows how to do cylindrical billboards
- sph_billboard.c - shows how to do spherical billboards
- warp.c - shows how to warp images with textures
- noise.c - shows how to make a filtered noise function
- spectral.c - shows how to make a spectral function from filtered noise
- spotnoise.c - shows how to use spot noise
- tex3dsolid.c - renders a solid image with a 3d texture
- tex3dfunc.c - creates a 2d texture that varies with r value
- makedetail.c - shows how to create a detail texture
- detail.c - shows how to use a detail texture
- aniso.c - shows how to create and use anisotropic textures
- cutaway.c - shows how to create a gradual cutaway

Line Rendering Techniques

- haloed.c - shows how to draw haloed lines using the depth buffer
- silhouette.c - shows how to draw the silhouette edge of an object with the stencil buffer
- solid_to_line.c - shows how to draw solid objects as lines
- overlap.c - shows how to draw wide, smoothed line loops with rounded edges

Blending and Compositing

- comp.c - shows Porter/Duff compositing
- transp.c - shows how to draw transparent objects
- imgproc.c - shows image processing operations
- transparent.c - shows transparency, ordering, culling interactions
- zcomposite.c - shows how to composite depth-buffered images

Antialiasing

- lineaa.c - shows how to draw antialiased lines
- texaa.c - shows how to antialias with texture
- accumaa.c - shows how to antialias a scene with the accumulation buffer
- aalines.c - more on antialiased lines
- aasolid.c - shows how to antialias solids

Lighting Techniques

- envphong.c - shows how to draw phong highlights with environment mapping
- lightmap2d.c - shows how to do 2D texture lightmaps
- lightmap3d.c - shows how to do 3D texture lightmaps
- bumpmap.c - shows how to bumpmap with texture
- fresnel.c - shows an example of how to render Fresnel reflections
- anisolight.c - shows an example of how to render anisotropic reflections

Scene Realism

- genspheremap.c - shows how to generate sphere maps
- mirror.c - shows how to do planar mirror reflections
- projshadow.c - shows how to render projection shadows
- shadowvol.c - shows how to render shadows with shadow volumes
- shadowmap.c - shows how to render shadows with shadow maps
- softshadow.c - shows how to do soft shadows with the accumulation buffer by jittering light sources
- softshadow2.c - shows how to do soft shadows by creating lighting textures with the accumulation buffer

Transparency

- screendoor.c - shows how to do screen-door transparency
- alphablend.c - shows how to do transparency with alpha blending

Image Processing

- convolve.c - shows how to convolve with the accumulation buffer
- cmatrix - shows how to modify colors with a color matrix

Special Effects

- dissolve.c - shows how to do dissolves with the stencil buffer
- motionblur.c - shows how to do motion blur with the accumulation buffer
- field.c - shows how to achieve depth of field effects with the accumulation buffer with the stencil buffer

Illustration and Artistic Techniques

- npr.c - shows how to implement a non-photorealistic lighting model
- hatch.c - shows how to do 3D cross-hatching
- 2d.c - shows how to do 2D rendering
- join.c - shows how to do various styles of line joins
- paint.c - shows how to generate an abstract image from a source image

Scientific Visualization Techniques

- plate.c - shows simple scalar field visualization
- vol2dtex.c - volume visualization with 2D textures
- vol3dtex.c - volume visualization with 3D textures
- lic.c - shows how to compute a line integral convolution
- illumline.c - shows how to use lit stream lines for vector field visualization

Natural Phenomena

- smoke.c - shows how to render smoke
- smoke3d.c - shows how to render 3D smoke using volumetric techniques
- vapor.c - shows how render a vapor trail
- texmovie.c - shows how to create a texture movie
- fire.c - shows how to animate fire
- explode.c - shows how to create an explosion
- dscloud.c - create a cloud image using diamond-square technique
- cloud.c - shows how to render a cloud layer
- cloudlayer.c - shows how to create ground fog
- cloud3d.c - shows how to render a 3D cloud using volumetric techniques

- fire.c - shows how to render fire using movie loops
- water.c - shows an example water rendering technique
- bubble.c - shows an example of how to render a bubble
- underwater.c - shows an example of rendering an underwater scene
- lightpoint.c - shows how to render point light sources
- particle.c - shows how to create particle systems
- snow.c - shows an example of rendering falling snow
- rain.c - shows an example of rendering falling rain

Application Tuning

- complexity.c - shows how to visualize depth complexity for a scene

B GLUT, the OpenGL Utility Toolkit

The example programs for these notes use ”GLUT”, a utility toolkit created by Mark Kilgard [55] and contributed to widely by the graphics community.

GLUT is easy to use and simple, so it may appeal to beginning OpenGL users. OpenGL users of all experience levels can use GLUT to rapidly prototype an algorithm using OpenGL and not spend time writing the code to configure an X Window, setting up a Win32 color map, etc.

The GLUT library provides a number of convenience functions for handling window systems and input devices. Applications can request an OpenGL visual using a set of attributes and manipulate the window that provides that visual through a window-system-independent API.

GLUT provides pop-up menu support and device handling support for a variety of devices such as keyboard, mouse, and trackball, and invokes user-supplied callbacks to handle window events such as exposure and resizing.

GLUT also offers utility routines for drawing several geometric shapes as solids or wireframe models, including spheres, tori, and teapots.

Text rendering is also simplified by GLUT. Several bitmap and stroke fonts are provided with the GLUT distribution.

GLUT is available on most UNIX platforms, MacOS, and Windows NT/95, and other operating systems. It can be downloaded from `http://www.opengl.org/Developers/Documentation/glut.html`.

C Equations

This section describes some important formula and matrices referred to in the text.

C.1 Projection Matrices

C.1.1 Perspective Projection

The call `glFrustum(l, r, b, t, n, f)` generates R, where:

$$R = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \text{ and } R^{-1} = \begin{pmatrix} \frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n} & \\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} & 0 \\ 0 & 0 & 0 & -1 & \\ 0 & 0 & \frac{-f-n}{2fn} & \frac{f+n}{2fn} & \end{pmatrix}$$

R is defined as long as $l \neq r$, $t \neq b$, and $n \neq f$.

C.1.2 Orthographic Projection

The call `glOrtho(l, r, b, t, u, f)` generates R, where:

$$R = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ and } R^{-1} = \begin{pmatrix} \frac{r-l}{2} & 0 & 0 & \frac{r+l}{2} \\ 0 & \frac{t-b}{2} & 0 & \frac{t+b}{2} \\ 0 & 0 & \frac{f-n}{2} & \frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

R is defined as long as $l \neq r$, $t \neq b$, and $n \neq f$.

C.1.3 Perspective z-Coordinate Transformations

The z value in eye coordinates, z_{eye}, can be computed from the window coordinate z value, z_{window}, using the near and far plane values, $near$ and far, from the `glFrustum` command and the viewport near and far values, far_{vp} and $near_{vp}$, from the `glDepthRange` command using the equation:

$$z_{eye} = \frac{\frac{far\,near(far_{vp} - near_{vp})}{far - near}}{z_{window} - \frac{(far + near)(far_{vp} - near_{vp})}{2(far - near)} - \frac{far_{vp} + near_{vp}}{2}}$$

The z window coordinate is computed from the eye coordinate z using the equation:

$$z_{window} = \left[\frac{far + near}{far - near} + \frac{2\,far\,near}{z_{eye}(far - near)}\right]\left[\frac{far_{vp} - near_{vp}}{2}\right] + \frac{far_{vp} + near_{vp}}{2}$$

C.2 Lighting Equations

C.2.1 Attenuation Factor

The attenuation factor is defined to be:

$$\text{attenuation factor } = \frac{1}{k_c + k_l d + k_q d^2}$$

where

$d =$ distance between the light's position and the vertex

$k_c =$ `GL_CONSTANT_ATTENUATION`

$k_l =$ `GL_LINEAR_ATTENUATION`

$k_q =$ `GL_QUADRATIC_ATTENUATION`

If the light is directional, the attenuation factor is 1.

C.2.2 Spotlight Effect

The *spotlight effect* evaluates to one of three possible values, depending on whether the light is actually a spotlight and whether the vertex lies inside or outside the cone of illumination produced by the spotlight:

- 1 if the light isn't a spotlight (`GL_SPOT_CUTOFF` is 180.0).
- 0 if the light is a spotlight but the vertex lies outside the cone of illumination produced by the spotlight.
- $(\max\{v \cdot d, 0\})^{GL\_SPOT\_EXPONENT}$ where: $v = (v_x, v_y, v_z)$ is the unit vector that points from the spotlight (`GL_POSITION`) to the vertex.

  $d = (d_x, d_y, d_z)$ is the spotlight's direction (`GL_SPOT_DIRECTION`), assuming the light is a spotlight and the vertex lies inside the cone of illumination produced by the spotlight.

  The dot product of the two vectors v and d varies as the cosine of the angle between them; hence, objects directly in line get maximum illumination, and objects off the axis have their illumination drop as the cosine of the angle.

To determine whether a particular vertex lies within the cone of illumination, OpenGL evaluates $(\max\{\hat{v} \cdot \hat{d}, 0\})$ where $\hat{v}$ and $\hat{d}$ are as defined above. If this value is less than the cosine of the spotlight's cutoff angle (`GL_SPOT_CUTOFF`), then the vertex lies outside the cone; otherwise, it's inside the cone.

C.2.3 Ambient Term

The ambient term is simply the ambient color of the light scaled by the ambient material property:

$$\text{ambient}_{light} * \text{ambient}_{material}$$

C.2.4 Diffuse Term

The diffuse term needs to take into account whether light falls directly on the vertex, the diffuse color of the light, and the diffuse material property:

$(\max\{l \cdot n, 0\}) * \text{diffuse}_{light} * \text{diffuse}_{material}$

where:

$l = (l_x, l_y, l_z)$ is the unit vector that points from the vertex to the light position (`GL_POSITION`).

$n = (n_x, n_y, n_z)$ is the unit normal vector at the vertex.

C.2.5 Specular Term

The specular term also depends on whether light falls directly on the vertex. If $\vec{l} \cdot \vec{n}$ is less than or equal to zero, there is no specular component at the vertex. (If it's less than zero, the light is on the wrong side of the surface.) If there's a specular component, it depends on the following:

- The unit normal vector at the vertex (n_x, n_y, n_z).
- The sum of the two unit vectors that point between (1) the vertex and the light position and (2) the vertex and the viewpoint (assuming that GL_LIGHT_MODEL_LOCAL_VIEWER is true; if it's not true, the vector $(0, 0, 1)$ is used as the second vector in the sum). This vector sum is normalized (by dividing each component by the magnitude of the vector) to yield $s = (s_x, s_y, s_z)$.
- The specular exponent (GL_SHININESS).
- The specular color of the light (GL_SPECULAR$_{light}$).
- The specular property of the material (GL_SPECULAR$_{material}$).

Using these definitions, here's how OpenGL calculates the specular term:

$$(\max\{s \cdot n, 0\})^{shininess} * \text{specular}_{light} * \text{specular}_{material}$$

However, if $\vec{l} \cdot \vec{n} = 0$, the specular term is 0.

C.2.6 Putting It All Together

Using the definitions of terms described in the preceding paragraphs, the following represents the entire lighting calculation in RGBA mode.

$$\begin{aligned}
\text{vertex color} \quad = \quad & \text{emission}_{material} + \\
& \text{ambient}_{lightmodel} * \text{ambient}_{material} + \\
& \sum_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) (\text{spotlight effect})_i \\
& (\text{ambient}_{light} * \text{ambient}_{material} + \\
& (\max\{l \cdot n, 0\}) * \text{diffuse}_{light} * \text{diffuse}_{material} + \\
& (\max\{s \cdot n, 0\})^{shininess} * \text{specular}_{light} * \text{specular}_{material})_i
\end{aligned}$$

References

[1] J. Airey, B. Cabral, and M. Peercy. Explanation of bump mapping with texture. Personal Communication, 1997.

[2] K. Akeley. The hidden charms of z-buffer. *Iris Universe*, (11):31–37, 1990.

[3] K. Akeley. OpenGL philosophy and the philosopher's drinking song. Personal Communication, 1996.

[4] K. Akeley. Algorithm for drawing boundary plus silhouette edges for a solid. Personal Communication, 1998.

[5] Y. Attarwala. Rendering hidden lines. *Iris Universe*, Fall:39, 1988.

[6] Y. Attarwala and M. Kong. Picking from the picked few. *Iris Universe*, Summer:40–41, 1989.

[7] Michael Bailey and Dru Clark. Encoding 3d surface information in a texture vector. *Journal of Graphics Tools*, 2(3):29–35, 1997. http://www.sdsc.edu/tmf/texvec.pdf.

[8] P. Bergeron. A general version of crow's shadow volumes. *IEEE Computer Graphics and Applications*, 6(9):17–28, 1986.

[9] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 286–292, August 1978.

[10] Jim Blinn. Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, January 1988. reprinted in the book *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, 1996.

[11] Jim Blinn and Martin Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19:456–547, 1976. Reprinted in *Tutorial: Computer Graphics*, 2nd ed., Editors John Beatty and Kellogg Booth, IEEE Computer Society, 1982.

[12] OpenGL Architecture Review Board. *OpenGL Reference Manual*. Addison-Wesley, second edition edition, 1997. ISBN 0-201-46140-4.

[13] A. Bourgoyne, R. Bornstein, and D. Yu. *Silicon Graphics Visual Workstation OpenGL Programming Guide For Windows NT*. Silicon Graphics, Mountain View, CA, 1999. https://www.sgi.com/developers/nt/sdk/.

[14] Lynee Shapiro Brotman and Norman Badler. Generating soft shadows with a depth buffer algorithm. *IEEE Computer Graphics and Applications*, October 1984.

[15] Jim Bushnell and Jason Mitchell. Advanced multitexture effects with direct3d and opengl. In *Game Developers Conference Proceedings 99*, pages 81–99, March 1999.

[16] Brian Cabral and Leith (Casey) Leedom. Imaging vector fields using line integral convolution. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 263–272, August 1993.

[17] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Harcourt Brace & Company, 1993.

[18] The VRML Consortium. The virtual reality modeling language specification. web site, August 1996. http://vag.vrml.org.

[19] S. Coorg and S. Teller. A spatially and temporally coherent object space visibility algorithm. Technical Report TM 546, Laboratory for Computer Science, Massachusetts Institute of Technology, 1996.

[20] F. C. Crow. A comparison of antialiasing techniques. *IEEE Computer Graphics and Applications*, 1(1):40–48, January 1981.

[21] J. D. Cutnell and K. W. Johnson. *Physics*. John Wiley & Sons, 1989.

[22] Michael F. Deering. High resolution virtual reality. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 195–202, July 1992.

[23] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 65–74, August 1988.

[24] Tom Duff. Compositing 3-D rendered images. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 41–44, July 1985.

[25] David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, October 1994. ISBN 0-12-228760-6.

[26] Francine Evans, Steven Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In *Proceedings of Visualization 96*, pages 319–326, 1996. http://www.cs.sunysb.edu/ evans/stripe.html.

[27] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Company, 1990.

[28] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. *Introduction to Computer Graphics*. Addison-Wesley Publishing Company, 1994.

[29] A. Fournier, D. Fussell, and L. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, June 1982.

[30] Alain Fournier and William T. Reeves. A simple model of ocean waves. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 75–84, August 1986.

[31] Geoffrey Y. Gardner. Visual simulation of clouds. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 297–303, July 1985.

[32] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufman Publishers, Inc., 1995.

[33] Jack Goldfeather, Jeff P. M. Hultquist, and Henry Fuchs. Fast constructive-solid geometry display in the Pixel-Powers graphics system. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 107–116, August 1986.

[34] Ronald Goldman. Matrices and transformations. In Andrew Glassner, editor, *Graphics Gems*, page 474. Academic Press, 1990.

[35] Rafael C. Gonzalez and Paul Wintz. *Digital Image Processing (2nd Ed.)*. Addison-Wesley, Reading, MA, 1987.

[36] A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A non-photorealistic lighting model for automatic technical illustration. In Michael F. Cohen, editor, *Computer Graphics (SIGGRAPH '98 Proceedings)*, volume 25, pages 447–452, July 1998.

[37] B. Gooch, P. Sloan, A. Gooch, P. SHirley, and R. Riesenfield. Interactive technical illustration. In J. Hodgins and J. Foley, editors, *Proceedings of the 1999 symposium on Interactive 3D Graphics*, pages 31–38, April 1999.

[38] H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, C-20(6):623–629, June 1971.

[39] P. Haeberli. Matrix operations for image processing. web site, November 1993. http://www.sgi.com/grafica/matrix/index.html.

[40] P. Haeberli and D. Voorhies. Image processing by linear interpolation and extrapolation. *Iris Universe*, (28):8–9, 1994.

[41] Paul Haeberli and Mark Segal. Texture mapping as a fundamental drawing primitive. In Michael F. Cohen, Claude Puech, and Francois Sillion, editors, *Fourth Eurographics Workshop on Rendering*, pages 259–266. Eurographics, June 1993. held in Paris, France, 14–16 June 1993.

[42] Paul E. Haeberli. Paint by numbers: Abstract image representations. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 207–214, August 1990.

[43] Paul E. Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 309–318, August 1990.

[44] Peter M. Hall and Alan H. Watt. Rapid volume rendering using a boundary-fill guided ray cast algorithm. In N. M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena (Proceedings of CG International '91)*, pages 235–249. Springer-Verlag, 1991.

[45] Roy Hall. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York, 1989. includes C code for radiosity algorithms.

[46] Pat Hanrahan and Paul E. Haeberli. Direct WYSIWYG painting and texturing on 3D shapes. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 215–223, August 1990.

[47] Paul S. Heckbert and Michael Herf. Fast soft shadows. In *Visual Proceedings, SIGGRAPH 96*, page 145. ACM Press, 1996. ISBN 0-89791-784-7.

[48] Paul S. Heckbert and Michael Herf. Shadow generation algorithms. web site, April 1997. http://www.cs.cmu.edu/ ph/shadow.html.

[49] T. Heidmann. Real shadows real time. *Iris Universe*, (18):28–31, 1991.

[50] Wolfgang Heidrich and Hans-Peter Seidel. Efficient rendering of anisotropic surfaces using computer graphics hardware. In *Image and Multi-dimensional Digital Signal Processing Workshop (IMDSP)*, 1998.

[51] Wolfgang Heidrich and Hans-Peter Seidel. View-independent environment maps. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1998. http://www9.informatik.uni-erlangen.de/eng/research/rendering/envmap.

[52] Russ Herrell, Joe Baldwin, and Chris Wilcox. High quality polygon edging. *IEEE Computer Graphics and Applications*, 15(4):68–74, July 1995.

[53] Tobias Huttner. High resolution textures. In *Proceedings of IEEE Visualization 98*, October 1998. http://davinci.informatik.uni-kl.de/vis98/archive/lbht/papers/huettnerA4.pdf.

[54] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 49–57, August 1990.

[55] Mark J. Kilgard. *Programming OpenGL for the X Window System*. Addison-Wesley, 1996. ISBN 0-201-48359-9.

[56] Mark J. Kilgard. Realizing opengl: Two implementations of one architecture. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1997. http://reality.sgi.com/mjk/twoimps/twoimps.html.

[57] Mark J. Kilgard. A simple opengl-based api for texture mapped text. web site, 1997. http://reality.sgi.com/opengl/tips/TexFont/TexFont.html.

[58] John-Peter Lewis. Algorithms for solid noise synthesis. In Jeffrey Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 263–270, July 1989.

[59] Terence Lindgren and John Weber. Measuring the quality of antialiased line drawing algorithms. In Michael F. Cohen, Claude Puech, and Francois Sillion, editors, *Fourth Eurographics Workshop on Rendering*, pages 157–174. Eurographics, June 1993. held in Paris, France, 14–16 June 1993.

[60] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings of the 1995 symposium on Interactive 3D Graphics*, page 105, 1995.

[61] Kwan-Liu Ma, Brian Cabral, Hans-Christian Hege, Detlev Stalling, and Victoria L. Interrante. *Texture Synthesis with Line Integral Convolution*. ACM SIGGRAPH, Los Angeles, 1997. Siggraph '97 Conference Course Notes.

[62] L. Markosian, M. Kowalski, S. Trychin, L. Bourdev, Goldstein D, and J. Hughes. Real-time nonphotorealistic rendering. In Turner Whitted, editor, *Computer Graphics (SIGGRAPH '97 Proceedings)*, volume 24, pages 415–420, August 1997.

[63] Gavin S. P. Miller. The definition and rendering of terrain maps. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 39–48, August 1986.

[64] Don P. Mitchell and Arun N. Netravali. Reconstruction filters in computer graphics. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 221–228, August 1988.

[65] John Montrym, Dan Baum, Dave Dignam, and Chris Migdal. Infinitereality: A real-time graphics system. In Turner Whitted, editor, *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 293–302, August 1997.

[66] H. R. Myler and A. R. Weeks. *The Pocket Handbook of Image Processing Algorithms in C*. University of Central Florida Department of Electrical & Computer Engineering, 1993.

[67] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley, second edition edition, 1997. ISBN 0-201-46138-2.

[68] Scott R. Nelson. Twelve characteristics of correct antialiased lines. *Journal of Graphics Tools*, 1(4):1–20, 1996.

[69] Scott R. Nelson. High quality hardware line antialiasing. *Journal of Graphics Tools*, 2(1):29–46, 1997.

[70] Tomoyuki Nishita and Eihachiro Nakamae. Method of displaying optical effects within water using accumulation buffer. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 373–381. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[71] Eyal Ofek. *Modeling and Rendering 3-D Objects*. PhD thesis, Institute of Computer Science, The Hebrew University, 1998.

[72] Eyal Ofek and Ari Rappoport. Interactive reflections on curved objects. In Michael F. Cohen, editor, *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 333–342, July 1998.

[73] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1994.

[74] Hewlett Packard. Opengl implementation guide. web site, June 1998. http://www.hp.com/unixwork/products/grfx/OpenGL/Web/ImpGuide.html.

[75] Darwyn R. Peachey. Modeling waves and surf. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 65–74, August 1986.

[76] M. Peercy. Explanation of sphere mapping. Personal Communication, 1997.

[77] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, 1997.

[78] Bui-T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[79] Thomas Porter and Tom Duff. Compositing digital images. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259, July 1984.

[80] Franco P. Preparata and Michael Ian Shamos. *Computation Geometry*. Springer-Verlag, New York, 1985.

[81] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 283–291, July 1987.

[82] David F. Rogers. *Procedural Elements for Computer Graphics*. McGraw-Hill, second edition edition, 1997.

[83] John Rohlf and James Helman. IRIS performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[84] P. Rustagi. Silhouette line display from shaded models. *Iris Universe*, Fall:42–44, 1989.

[85] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-D shapes. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 197–206, August 1990.

[86] John Schlag. *Fast Embossing Effects on Raster Image Data*. Academic Press, Cambridge, 1994.

[87] M. Schulman. Rotation alternatives. *Iris Universe*, Spring:39, 1989.

[88] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, Inc., Mountain View, CA, October 1998. includes the ARB_multitexture specification; ftp://sgigate.sgi.com/pub/opengl/doc/opengl1.2/opengl1.2.1.pdf.

[89] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul E. Haeberli. Fast shadows and lighting effects using texture mapping. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 249–252, July 1992.

[90] Peter-Pike J. Sloan, David Weinstein, and J. Dean Brederson. Importance driven texture coordinate optimization. Submitted to SIGGRAPH '97, 1997. http://www.cs.utah.edu/ dejohnso/workshop/talks/sloan/sloan.html.

[91] Cyril Soler and Francois Sillion. The clipmap: A virtual mipmap. In Michael F. Cohen, editor, *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 321–332, July 1998.

[92] D. Stalling, M. Zockler, and H.-C. Hege. Fast display of illuminated field lines. In *IEEE Transactions on Visualization and Computer Graphics*, volume 3, pages 118–128, 1997.

[93] Paul S. Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 341–349, July 1992.

[94] Chris Tanner, Chris Migdal, and Michael Jones. The clipmap: A virtual mipmap. In Michael F. Cohen, editor, *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 151–158, July 1998.

[95] M. Teschner. Texture mapping: New dimensions in scientific and technical visualization. *Iris Universe*, (29):8–11, 1994.

[96] T. Tessman. Casting shadows on flat surfaces. *Iris Universe*, Winter:16, 1989.

[97] Jarke J. van Wijk. Spot noise-texture synthesis for data visualization. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 309–318, July 1991.

[98] Douglas Voorhies and Jim Foran. Reflection vector shading hardware. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 163–166. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[99] Bruce Walter, Gun Alppay, Eric Lafortune, Sebastian Fernandez, and Donald P. Greenberg. Fitting virtual lights for non-diffuse walkthroughs. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, volume 31, pages 45–48, August 1997.

[100] Mark Watt. Light-water interaction using backward beam tracing. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 377–385, August 1990.

[101] T. F. Wiegand. Interactive rendering of csg models. In *Computer Graphics Forum*, volume 15, pages 249–261, 1996.

[102] Tim Wiegand. Cadlab open inventor node library: csg. web site, April 1998. http://www.arct.cam.ac.uk/research/cadlab/inventor/csg.html.

[103] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 1–11, July 1983.

[104] Andrew Woo, Pierre Poulin, and Alain Fournier. A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, November 1990.

[105] H. Zhang, D. Manocha, T. Hudson, and K. Hoff III. Visibility culling using hierarchical occlusion maps. In Turner Whitted, editor, *Computer Graphics (SIGGRAPH '97 Proceedings)*, volume 24, pages 77–88, August 1997.

[106] Hansong Zhang. Effective occlusion culling for the interactive display of arbitrary models. Doctoral dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, 1998. http://www.cs.unc.edu/ zhangh/dissertation.pdf.

Volume 15, (1997) number 4 pp. 249–261

Interactive Rendering of CSG Models

T. F. Wiegand[†]

The Martin Centre for Architectural and Urban Studies
The University of Cambridge, Cambridge, UK

Abstract
We describe a CSG rendering algorithm that requires no evaluation of the CSG tree beyond normalization and pruning. It renders directly from the normalized CSG tree and primitives described (to the graphics system) by their facetted boundaries. It behaves correctly in the presence of user defined, "near" and "far" clipping planes. It has been implemented on standard graphics workstations using Iris GL [?] and OpenGL [?] graphics libraries. Modestly sized models can be evaluated and rendered at interactive (less than a second per frame) speeds. We have combined the algorithm with an existing B-rep based modeller to provide interactive rendering of incremental updates to large models.

1. Introduction

Constructive Solid Geometry (CSG) within an interactive modelling environment provides a simple and intuitive approach to solid modelling. In conventional modelling systems primitives are first positioned, a boolean operation is performed and the results then rendered. Often the correct position cannot be gauged easily from display of the primitives alone. A sequence of trial and error may be initiated or perhaps a break from the normal modelling process to calculate the correct position numerically. Conceptual modelling is inhibited — usually a design is fully fledged before modelling commences. Interactive rendering offers the promise of a modelling system where designers can easily explore possibilities within the CSG paradigm. For instance, a designer could drag a hole defined by a complex solid through a workpiece, observing the new forms that emerge.

Interactive rendering of CSG models has previously been implemented with special purpose hardware [?, ?, ?]. We believe that such systems should be based on an existing, commonly available graphics library. Use of an existing graphics library simplifies development, protects investment in proprietary graphics hardware, and leverages off future improvements in the hardware supported by the library. Conversion of the CSG tree for a model into a boundary representation (B-rep) meets this goal but is typically too slow for interactive modification.

The surfaces in the B-rep of a model are a subset of the surfaces of the primitives in the CSG tree for the model. Conversion to a B-rep is then the classification of the surfaces of each primitive into portions that are "inside", "outside", or "on" the surface of the fully evaluated model. Display of the model only requires classification of the points on the surfaces which project to each pixel. Point classification is much simpler than surface classification. Geometrically, point classification requires intersection of the primitives with rays through each pixel, while surface classification requires intersection of the primitive surfaces with each other.

Thibault and Naylor [?] describe a surface classification based approach. They build BSP trees for each primitive and perform the classification by merging the trees together. The resulting tree is equivalent to a BSP tree built from the B-rep of the model. The complete evaluation process is too slow for interactive rendering. They describe an incremental version of their algorithm which provides interactive rendering speeds within a modelling environment.

There are variations of most rendering algorithms which use point classification. These include ray trac-

[†] Supported by Informatix, Inc. Tokyo.

ing ?, scan line methods ?, and depth-buffer methods ?, ?, ?. Much attention has been focused on optimising point classification for this purpose ?. These algorithms all add point classification within the lowest levels of the standard algorithms. We require an algorithm which can be implemented using an existing graphics library.

Goldfeather, Molnar, Turk and Fuchs ? describe an algorithm that first normalizes a CSG tree before rendering the normalized form. It operates in a SIMD pixel parallel way on an augmented frame buffer (Pixel-planes 4) which has two depth (Z) buffers, two color buffers and flag bits per pixel. We have developed a new version of this algorithm capable of being implemented using an existing graphics library on a conventional graphics workstation. Our algorithm requires a single depth buffer, single color buffer, stencil (flag bits) buffer and the ability to save and restore the contents of the depth buffer.

In section **??** we review the algorithm described by Goldfeather et. al. ?. We have restructured the presentation of the ideas to make them more amenable to implementation on a conventional graphics workstation. Our implementation is described in section **??**. In section **??** we describe the integration of user defined, "near" and "far" clipping planes into the algorithm. In section **??** we describe use of the algorithm within an interactive modelling system. The system maintains fully evaluated B-rep versions of models and uses the rendering algorithm for interactive changes to the models. Section **??** presents performance statistics for our current implementation using the Silicon Graphics GL library ?.

2. Rendering a CSG tree using pixel parallel operations

We would advise interested readers to refer to Goldfeather et. al. ? for a fuller description of the algorithm which we summarize in this section.

A CSG tree is either a primitive or a boolean combination of sub-trees with intersection($\cap$), subtraction($-$) or union($\cup$) operators. A CSG tree is in normal (sum of products) form when all intersection or subtraction operators have a left subtree which contains no union operators and a right subtree that is simply a primitive. For example $(((A\cap B)-C)\cup(D\cap(E-(F\cap G))))\cup H$, where A–H represent primitives, is in normal form. We shall assume left association of operators so the previous expression can be written as $(A\cap B-C)\cup(D\cap E-F\cap G)\cup H$. This expression has three products. The primitives A, B, D, E, G, H are *uncomplemented*, C and F are *complemented.*

The normalization process recursively applies a set of production rules to a CSG tree which use the associative and distributive properties of boolean operations. Determining an appropriate rule and applying it uses only local information (type of current node and child node types). The production rules and algorithm used are :

1. $X-(Y\cup Z) \rightarrow (X-Y)-Z$
2. $X\cap(Y\cup Z) \rightarrow (X\cap Y)\cup(X\cap Z)$
3. $X-(Y\cap Z) \rightarrow (X-Y)\cup(X-Z)$
4. $X\cap(Y\cap Z) \rightarrow (X\cap Y)\cap Z$
5. $X-(Y-Z) \rightarrow (X-Y)\cup(X\cap Z)$
6. $X\cap(Y-Z) \rightarrow (X\cap Y)-Z$
7. $(X-Y)\cap Z \rightarrow (X\cap Z)-Y$
8. $(X\cup Y)-Z \rightarrow (X-Z)\cup(Y-Z)$
9. $(X\cup Y)\cap Z \rightarrow (X\cap Z)\cup(Y\cap Z)$

```
proc normalize($T$ : tree)
{
    if $T$ is a primitive {
      return
    }
    repeat {
            while $T$ matches a rule from 1–9 {
                    apply first matching rule
            }
            normalize($T$.left)
    } until ($T$.op is a union) or
            (($T$.right is a primitive) and
             ($T$.left is not a union))
    normalize($T$.right)
}
```

Goldfeather et. al. ? show that the algorithm terminates, generates a tree in normal form and does not add redundant product terms or repeat primitives within a product.

Normalization can add many primitive leaf nodes to a tree with a possibly exponential increase in tree size. In most cases, a large number of the products generated by normalization play no part in the final image, because their primitives do not intersect. A limited amount of geometric information (bounding boxes of primitives) is used to prune CSG trees as they are normalized. Bounding boxes are computed for each operator node using the rules :

1. $\text{Bound}(A\cup B) = \text{Bound}(\text{Bound}(A)\cup\text{Bound}(B))$
2. $\text{Bound}(A\cap B) = \text{Bound}(\text{Bound}(A)\cap\text{Bound}(B))$
3. $\text{Bound}(A-B) = \text{Bound}(A)$

Here A and B are arbitrary child nodes. After each step of the normalization algorithm the tree is pruned by applying the following rules to the current node :

1. $A \cap B \rightarrow \emptyset$, if Bound($A$) does not intersect Bound(B).
2. $A - B \rightarrow A$, if Bound(A) does not intersect Bound(B).

Normalization of the tree allows simplification of the rendering problem. The union of two or more solids can be rendered using the standard depth (Z) buffer hidden surface removal algorithm used by most graphics workstations. The rendering algorithm needs only to render the correct depth and color for each product in the normalized CSG tree and then allow the depth buffer to combine the results for each product.

Each product can be rendered by rendering each visible surface of a primitive and trimming (intersecting or subtracting) the surface with the remaining primitives in the product. The visible surfaces are the front facing surfaces of uncomplemented primitives and the back facing surfaces of complemented primitives. This observation allows a further rewriting of the CSG tree where each product is split into a sum of *partial products*. A convex primitive has one pair of front and back surfaces per pixel. A non-convex primitive may have any number of pairs of front and back surfaces per pixel. A k-convex primitive is defined as one that has at most k pairs of front and back surfaces per pixel from any view point. We shall use the notation A_k to represent a k-convex primitive and A_{fn} to represent the nth front surface (numbered 0 to $k-1$) of primitive A_k and A_{bn} to represent the nth back surface of A_k. In the common case of convex primitives, we shall drop the numerical subscripts. Thus, $A{-}B$ expands to $(A_f{-}B)\cup(B_b{\cap}A)$ in sum of partial products form; while $A_2{-}B$ expands to $(A_{f0}{-}B)\cup(A_{f1}{-}B)\cup(B_b{\cap}A_2)$. We call the primitive whose surface is being rendered the *target* primitive of the partial product. The remaining primitives are called *trimming* primitives.

The sum of partial products form again simplifies the rendering problem. It is now reduced to correctly rendering partial products before combining the results with the depth buffer. Additional *difference* pruning may also be carried out when products have been expanded to partial products :

3 $A_b \cap B \rightarrow \emptyset$, if Bound($A$) does not intersect Bound(B).

A partial product is rendered by first rendering the target surface of the partial product. Each pixel in the surface is then classified in parallel against each of the trimming primitives. To be part of the partial product surface, each pixel must be *in* with respect to any uncomplemented primitives and *out* with respect to any complemented ones. Those pixels which do not meet these criteria are trimmed away (colour set to background, depth set to initial value).

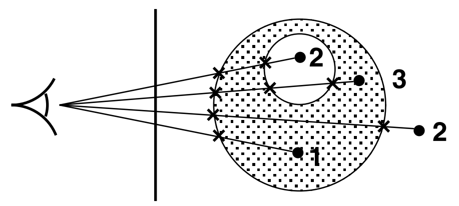


Figure 1: *Classifying per pixel depth values against a primitive*

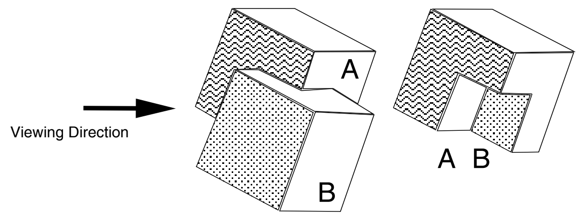


Figure 2: *A simple CSG expression*

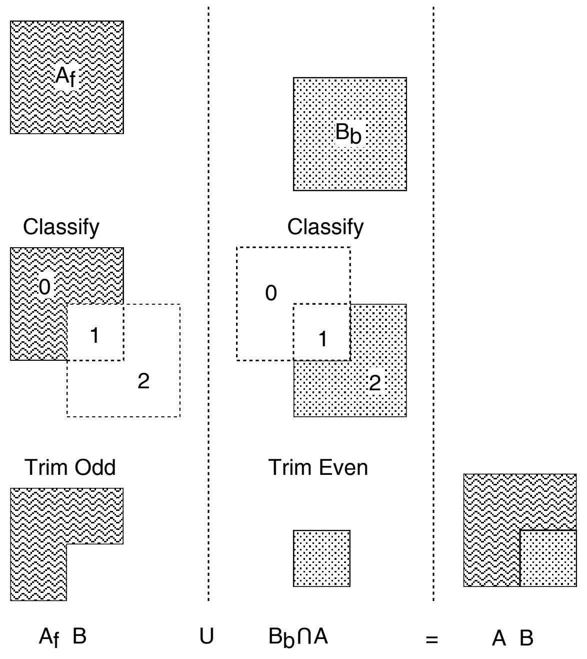


Figure 3: *Rendering figure ?? as two partial products*

Primitives must be formed from closed (possibly nested) facetted shells. Pixels can then be classified against a trimming primitive by counting the number of times a primitive fragment is closer during scan conversion of the primitive's faces. If the result is odd the pixel is *in* with respect to the primitive (figure ??). Pixels can be classified in parallel by using a 1 bit flag per pixel whose value is toggled whenever scan conversion of a trimming primitive fragment is closer than the pixel's depth value.

Figure ?? illustrates the process for $A-B$ looking along the view direction shown in figure ??. First, A_f is rendered, classified against B and trimmed (A_f-B). Then B_b is rendered, classified against A and trimmed ($B_b \cap A$). Finally, the two renders are composited together.

Rendering the appropriate surface of a convex primitive is simple as there is only one pair of front and back surfaces per pixel. Most graphics libraries support front and back face culling modes. To render all possible surfaces of an arbitrary k-convex primitive separately requires a $\log_2 k$ bit count per pixel. To render the jth front (or back) facing surface of a primitive, the front (or back) facing surfaces are rendered incrementing the count for each pixel and only enabling writes to the colour and depth buffers for which the count is equal to j.

3. Implementation on a conventional graphics workstation

The algorithm described in section ?? maps naturally onto a hardware architecture which can support two depth buffers, two colour buffers and a stencil buffer. One pair of depth and colour buffers, together with the stencil buffer, are used to render each partial product. The results are then composited into the other pair of buffers. Unfortunately, conventional graphics workstation hardware typically supports only one depth buffer. One approach is to use the hardware provided depth, colour and stencil buffers to render partial products; retrieving the results from the hardware and compositing in local workstation memory. The final result can then be returned direct to the frame buffer. This approach does not make the best use of the workstation hardware. Modern hardware tends to be highly pipelined. Interrupting the pipeline to retrieve results for each partial product will have a considerable performance penalty. In addition, the hardware is typically optimized for flow of data from local memory, through the pipeline and into the frame buffer. Data paths from the frame buffer back to local memory are likely to be slow, especially given the volume of data to be retrieved compared to the compact instructions given to the hardware to draw the primitives. Finally, the compositing operation in local memory will receive no help from the hardware.

Our approach attempts to extract the maximum benefit from any graphics hardware by minimizing the traffic between local memory and the hardware and by making sure that the hardware can be used for all rendering and compositing operations. The idea is to divide the rendering process into two phases — classification and final rendering. Before rendering begins the current depth buffer contents are saved into local memory. We then classify each partial product surface in turn. An extra stencil buffer bit (*accumulator*) per surface stores the results of the classification. During this process updates to the colour buffer are disabled. Once classification is complete, we restore the depth buffer to the saved state and enable updates to the colour buffer. Finally, each partial product surface is rendered again using the stored classification results as a mask (or stencil) to control update of the frame buffer. At the same time the depth buffer acts to composite the pixels which pass the stencil test with those already rendered.

The number of surfaces for which we can perform classification is limited by the depth of the stencil buffer. If the capacity of the stencil buffer is exceeded the surfaces must be processed in multiple passes with the depth buffer saved and restored during each pass. We can reduce the amount of data that needs to be copied by only saving the parts of the depth buffer that will be modified by classification during each pass. The first pass of each frame does not need to save the depth buffer at all as the values are known to be those produced by the initial clear. Instead of restoring, the depth buffer is cleared again. Thus, for simple models rendered at the start of a frame, no depth buffer save and restore is needed at all.

A surface may appear in more than one partial product in the normalized CSG tree. We exploit this by using the same accumulator bit for all partial products with the same surface. Classification results for each partial product are ORed with the current contents of the accumulator.

The stencil bits are partitioned into count bits (S_{count}), a parity bit (S_p) and an accumulator bit (S_a) per surface. $\log_2 k$ count bits are required where k is the maximum convexity of any primitive with a surface being classified in the current pass. The count and parity bits are used independently and may be overlapped. Table ?? shows the number of stencil buffer bits required to classify and render a single surface for primitives of varying convexity. The algorithm requires an absolute minimum of 2 bits for 1-convex and 2-convex primitives, classifying and rendering a single surface in a pass. In practice nearly all primitives used

Convexity	1	2	3–4	5–8	9–16	17-32	33-64	65–128
S_p	1	1	1	1	1	1	1	1
S_{count}	0	1	2	3	4	5	6	7
S_p and S_{count}	1	1	2	3	4	5	6	7
With 1 accumulator (S_0)	2	2	3	4	5	6	7	8
With 3 accumulators ($S_{0..2}$)	4	4	5	6	7	8	9	10
With 7 accumulators ($S_{0..6}$)	8	8	9	10	11	12	13	14

Table 1: *Stencil buffer usage with primitive convexity*

in pure CSG trees are 1-convex. With 8 stencil bits the algorithm can render from 7 1-convex primitives, to 1 surface of a 128-convex primitive, in a single pass.

Partial products are gathered into groups such that all the partial products in a group can be classified and rendered in one pass. The capacity of a group is defined as the number of different target surfaces that partial products in the group may contain. Capacity is dependent on the stencil buffer depth and the greatest convexity of any of the target primitives in the group (table **??**). Groups are formed by adding partial products in ascending order of target primitive convexity. Once one partial product with a particular target surface is added, all others with the same target surface can be added without using any extra capacity. Adding a partial product with a higher convexity than any already in the group will reduce the group capacity. If there is insufficient capacity to add the minimum convexity remaining partial product, a new group must be started.

Each group is processed in a separate pass in which all target surface primitives are classified and then rendered. Frame buffer wide operations are limited to areas defined by the projection of the bounding box of the current group or partial product. We present pseudo-code for the complete rendering process below. The procedures “glPrim(prim, tests, buffers, ops, pops)” and “glSet(value, tests, buffer, ops, pops)” should be provided by the graphics library. The first renders (scan converts) a primitive where “tests” are the tests performed at each pixel to determine if it can be updated, “buffers” specifies the set of buffers enabled for writing if the “tests” pass (where C is colour, Z is depth and S is stencil), “ops” are operations performed on the stencil bits at each pixel in the primitive, and “pops” are operations to be performed on the stencil bits at each pixel only if “tests” pass. The second procedure is similar but attempts to globally set values for all pixels. Iris GL ? and OpenGL ? are two graphics libraries which provide equivalents to the glPrim and glSet procedures described here. We use the symbol Z_P to denote the depth value at a pixel due to the scan conversion of a primitive, P. Hence, “$Z_P < Z$” is the familiar Z buffer hidden surface removal test. We use Z_f to represent the furthest possible depth value.

```
glSet(0, ALWAYS, $S$, $\emptyset$, $\emptyset$)
glSet(“far”, ALWAYS, $Z$, $\emptyset$, $\emptyset$)
for first group $G$ {
    classify($G$)
    glSet($Z_f$, ALWAYS, $Z$, $\emptyset$, $\emptyset$)
    renderGroup($G$)
} for each subsequent group $G$ {
      save depth buffer
      glSet($Z_f$, ALWAYS, $Z$, $\emptyset$, $\emptyset$)
      classify($G$)
      restore depth buffer
      renderGroup($G$)
}

proc classify($G$ : group)
{
      $a = 0$
      for each target surface $B$ in $G$ {
          for each partial product $R$ {
              renderSurface($B$)
              for each trimming primitive $P$ in $R$ {
                  trim($P$)
              }
              glSet(1, $S_a = 0$ & $Z \neq Z_f$, $S_a$, $\emptyset$, $\emptyset$)
              glSet($Z_f$, ALWAYS, $Z$, $\emptyset$, $\emptyset$)
          }
          $a = a + 1$
      }
}

proc renderGroup($G$ : group)
{
      $a = 0$
      for each target primitive $P$ in $G$ {
          glPrim($P$, $S_a = 1$ & $Z_P < Z$, $C$ & $Z$, $\emptyset$, $\emptyset$)
          glSet(0, ALWAYS, $S_a$, $\emptyset$, $\emptyset$)
          $a = a + 1$
      }
}
```

Capacity		Maximum Target Primitive Convexity							
		1	2	3–4	5–8	9–16	17-32	33-64	65–128
Stencil Buffer Depth	2	1	1	-	-	-	-	-	-
	3	2	2	1	-	-	-	-	-
	4	3	3	2	1	-	-	-	-
	5	4	4	3	2	1	-	-	-
	6	5	5	4	3	2	1	-	-
	7	6	6	5	4	3	2	1	-
	8	7	7	6	5	4	3	2	1

Table 2: *Group Capacity*

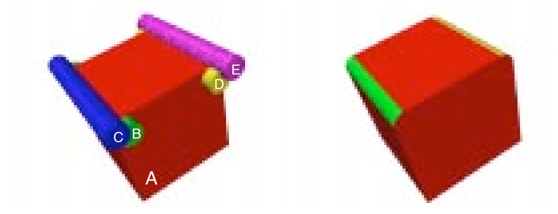


Figure 4: *(a) Primitives, (b) Rendering* $(A \cap B \cup A - C) \cap (A \cap D \cup A - E)$

```
proc renderSurface($B$ : surface)
{
        $P$ = target primitive containing $B$
        $n$ = surface number of $B$
        $k$ = convexity of $P$
        if $P$ is uncomplemented {
           enable back face culling
        } else {
           enable front face culling
        }
        if $k = 1$ {
           glPrim($P$, ALWAYS, $Z$, $\emptyset$, $\emptyset$)
        } else {
           glPrim($P$, $S_{count} = n$, $Z$, inc $S_{count}$, $\emptyset$)
           glSet(0, ALWAYS, $S_{count}$, $\emptyset$, $\emptyset$)
        }
}

proc trim($P$ : primitive)
{
        glPrim($P$, $Z_P < Z$, $\emptyset$, $\emptyset$, toggle $S_p$)
        if $P$ is uncomplemented {
           glSet($Z_f$, $S_p = 0$ , $Z$, $\emptyset$, $\emptyset$)
        } else {
           glSet($Z_f$, $S_p = 1$ , $Z$, $\emptyset$, $\emptyset$)
        }
        glSet(0, ALWAYS, $S_p$, $\emptyset$, $\emptyset$)
}
```

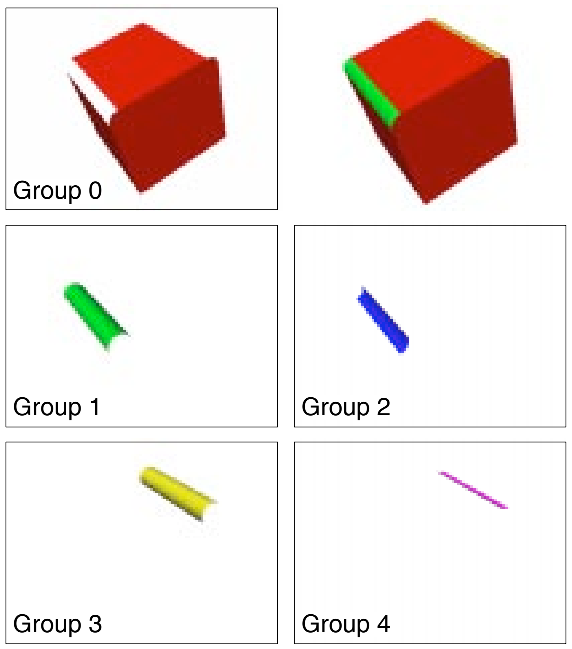


Figure 5: *Rendering each product group separately*

Figure **??** shows five primitives and a rendered CSG tree of the primitives. The expression $((A \cap B) \cup (A - C)) \cap ((A \cap D) \cup (A - E))$ normalizes to $(A \cap B \cap D) \cup (A \cap D - C) \cup (A \cap B - E) \cup (A - C - E)$. Expanding to partial products and grouping gives :

0: $(A_f \cap B \cap D) \cup (A_f \cap D - C) \cup (A_f \cap B - E) \cup (A_f - C - E)$
1: $(B_f \cap A \cap D) \cup (B_f \cap A - E)$
2: $(C_b \cap A \cap D) \cup (C_b \cap A - E)$
3: $(D_f \cap A \cap B) \cup (D_f \cap A - C)$
4: $(E_b \cap A \cap B) \cup (E_b \cap A - C)$

Figure **??** shows the result of rendering each product group separately. Product groups 2 and 4 are not visible in the combined image as they are behind the surfaces from groups 1 and 3.

4. Clipping planes and half spaces

Interactive inspection of solid models is aided by means of clipping planes which can help reveal internal structure. After a clipping plane has been defined and activated all subsequently rendered geometry is clipped against the plane and the parts on the *out* side discarded. The rendering of solids as closed shells means that clipping will erroneously reveal the interior of a shell when a portion of the shell is clipped away. Rossignac, Megahed and Schneider ? describe a stencil buffer based technique for "capping" shells where they intersect a clipping plane. Their algorithm will also highlight interferences (intersections) between solids on the clipping plane.

Clipping a solid and then capping is equivalent to intersection with a half space. We can trivially render an intersection between a solid S and a halfspace H by constructing a convex polygonal primitive P where one face lies on the plane defining H and has edges which do not intersect the bounding box of S. The other faces of P should not intersect S at all. Rendering $S \cap P$ is equivalent to rendering the solid defined by $S \cap H$.

Rossignac, Megahed and Schneider's ? capping algorithm can be easily integrated with our algorithm to make use of auxiliary clipping planes in rendering CSG trees involving halfspaces. As a halfspace is infinite we assume that it will always be intersected with a finite primitive in any CSG expression. Note that $S-H$ is equivalent to $S \cap \overline{H}$ where $\overline{H}$ is simply H with the normal of the halfspace defining plane reversed.

A halfspace acts as a trimming primitive by activating a clipping plane for the halfspace during the rendering of the target primitive. The stencil buffer is unused. The set of halfspaces in a product can be considered as a 1-convex target primitive. Its surface can be rendered by rendering the defining plane (or rather a sufficiently large polygon lying on the plane) of each halfspace while clipping planes are active for each of the other halfspaces. Each clipping plane is deactivated while it is being rendered to prevent it from clipping itself.

```
**proc** render($H$ : halfspace set)
{
    **for** each defining plane $P$ of $H$ {
        Activate clipping plane defined by $P$
    }
    **for** each front facing defining plane $P$ of $H$ {
        Deactivate clipping plane defined by $P$
        renderPlane($P$)
        Activate clipping plane defined by $P$
    }
    **for** each defining plane $P$ of $H$ {
        Deactivate clipping plane defined by $P$
    }
}
```

This approach has three advantages over rendering halfspaces as normal primitives. Firstly, the halfspace set only has to be rendered as a target primitive, all trimming by halfspaces uses the clipping planes. Secondly, each target primitive is clipped, reducing the amount of data written to the frame buffer at the cost of the extra geometry processing required by clipping. Thirdly, a solid/halfspace intersection can be correctly rendered using the algorithm for 1-convex solids ($k = 1$), independent of actual primitive convexity.

Rendering a k-convex target primitive using the algorithm for 1-convex solids results in the nearest surface being drawn (with depth buffering active). The nearest surface (*after* clipping) of a concave primitive will be visible in the intersection with a half space. Rendering an arbitrary CSG tree using the 1-convex algorithm will render the result of evaluating the CSG description on the "nearest spans" (nearest front to nearest back facing surface for each pixel) of each primitive. For interactive use the nearest spans are often all we are interested in. If not, then clipping planes may be used to delimit regions of interest within which the nearest spans will be correctly rendered. Thus, a lower cost, reduced quality mode of rendering is also available.

In addition to user defined clipping planes, all geometry is usually clipped to "near" and "far" planes. These planes are perpendicular to the viewing direction. All geometry must be further from the eye position than the near plane and nearer than the far plane. The near and far planes also define the mapping of distances from the eye point to values stored in the depth buffer. Points on the near plane map to the minimum depth buffer value and points on the far plane map to the maximum depth buffer value. The algorithm described in section **??** will fail if any primitive is clipped by either the near or far clipping plane.

In practice the far clipping plane can always be safely positioned beyond the primitives. The near plane is more troublesome. Firstly, it cannot be positioned behind the eye point. Secondly, the resolution of the depth buffer is critically dependent on the position of the near clip plane. It should be positioned as far from the eye point as possible. Consider rendering $A-B$ and positioning the eye in the hole in A formed by subtracting B. Near plane clipping is unavoidable. We can extend our algorithm to cap trimming prim-

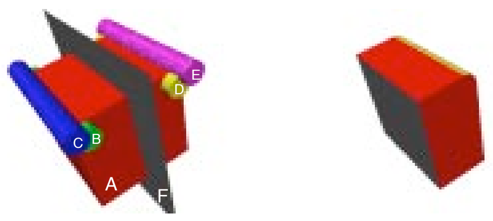


Figure 6: *(a) Primitives, (b) Rendering* $(A \cap B \cup A - C) \cap (A \cap D \cup A - E) \cap F$

itives if they will be subject to near plane clipping. Clipping of target primitives is not a problem unless the eye point is positioned inside the evaluated CSG model.

The trimming primitive is rendered twice while toggling S_p; firstly, with the depth buffer test disabled; secondly, with the depth buffer test enabled. The first render sets the parity bit where capping is required. The second completes the classification as above.

```
proc trim($P$ : primitive)
{
        glPrim($P$, ALWAYS, $\emptyset$, $\emptyset$, toggle $S_p$)
        glPrim($P$, $Z_P < Z$, $\emptyset$, $\emptyset$, toggle $S_p$)
        if $P$ is uncomplemented {
            glSet($Z_f$, $S_p = 0$ , $Z$, $\emptyset$, $\emptyset$)
        } else {
            glSet($Z_f$, $S_p = 1$ , $Z$, $\emptyset$, $\emptyset$)
        }
        glSet(0, ALWAYS, $S_p$, $\emptyset$, $\emptyset$)
}
```

Figure **??** shows our earlier example intersected with a single clipping plane / half space. The normalized CSG description is $(A \cap B \cap D \cap F) \cup (A \cap D \cap F - C) \cup (A \cap B \cap F - E) \cup (A \cap F - C - E)$.

The normalization and pruning algorithm described in section **??** needs to be extended to cope with half-space primitives. The extensions required are in the form of additional rules for bounding box generation, normalization and pruning (H is a halfspace) :

Bounding Box Generation

4. Bound$(A \cap H)$ = Bound(A)

Normalization

0. $X - H \rightarrow X \cap \overline{H}$

Pruning

4. $A \cap H$	$\rightarrow \emptyset$,	if Bound(A) is outside H.
5. $A \cap H$	$\rightarrow A$,	if Bound(A) is inside H.
6. $A \cap H - B$	$\rightarrow A \cap H$,	if Bound(B) is outside H.
7. $A_b \cap H$	$\rightarrow A_b$,	if Bound(A) is inside H.
8. $H_f - A$	$\rightarrow H_f$,	if Bound(A) does not intersect H.

Our earlier example (figure **??**) contains many pruning possibilities. The normalized CSG tree is $(A \cap B \cap D \cap F) \cup (A \cap D \cap F - C) \cup (A \cap B \cap F - E) \cup (A \cap F - C - E)$. Using rule 1 removes the product $A \cap B \cap D \cap F$ as B and D don't intersect. Rule 2 will reduce the products $A \cap D \cap F - C$ and $A \cap B \cap F - E$ to $A \cap D \cap F$ and $A \cap B \cap F$ as the complemented primitives do not intersect the product. Rule 4 removes the product $A \cap B \cap F$, rule 5 reduces $A \cap D \cap F$ to $A \cap D$ and rule 6 reduces $A \cap F - C - E$ to $A \cap F - E$. The normalized and geometric pruned CSG tree is then $(A \cap D \cap F) \cup (A \cap F - E)$. Expanding to partial products gives $(A_f \cap D \cap F) \cup (D_f \cap A \cap F) \cup (F_f \cap A \cap D) \cup (A_f \cap F - E) \cup (F_f \cap A - E) \cup (E_b \cap A \cap F)$. Finally, difference pruning will reduce $E_b \cap A \cap F$ to $E_b \cap A$ (rule 7) and $F_f \cap A - E$ to $F_f \cap A$ (rule 8).

We also prune products against the viewing volume for the current frame and classify trimming primitive bounding boxes against the near clipping plane to determine whether the extra capping step is necessary.

5. Interactive Rendering

We have incorporated our rendering algorithm in a simple, interactive solid modelling system built with standard components. The main framework is provided by the Inventor object-oriented 3D toolkit [?]. A model is represented by a directed acyclic graph of *nodes*. Operations on models, such as rendering or picking, are performed by means of *actions*. The toolkit may be extended by providing user written nodes and actions. Conventional solid modelling operations are provided by the ACIS geometric modeller [?]. ACIS is an object-oriented, boundary representation, solid modelling kernel.

Our modelling system adds new node types to Inventor which support ACIS modelled solids and CSG trees of solids. We also add a new rendering action which uses our stencil buffer CSG display algorithm to render CSG trees described by Inventor node graphs. A CSG evaluate action uses ACIS to fully evaluate a CSG tree allowing the tree to be replaced with a single evaluated solid node. All the standard Inventor interactive tools are available for editing models.

The system supports large CSG trees while maintaining interactive rendering speeds. During display and editing of a large CSG tree, only a small part of the model will be changing at any time. We "cache"

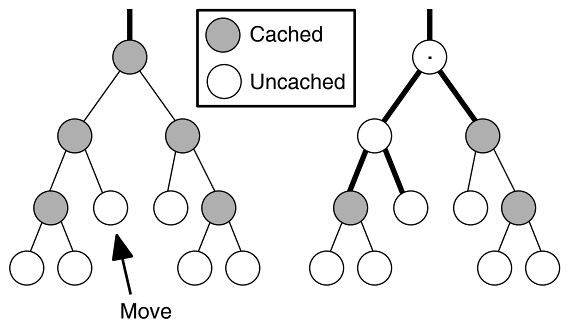


Figure 7: *Direct rendering of a CSG tree with cached geometry : (a) all caches valid, (b) limited direct rendering when a primitive is moved*

fully evaluated geometry obtained from the solid modeller at each internal node in the CSG tree. As caches become invalidated through editing of the model, portions of the tree are rendered directly (see figure ??), while the cached geometry is re-evaluated in the background (possibly on other workstations in a common network).

Current use of the system follows a common pattern. A user will quickly position and combine primitives using the solid modelling capabilities. During this stage the model is simple enough for the user to envisage the CSG operations required and to position primitives correctly. Figure ?? shows an example model of two intersecting corridors. Firstly, the space occupied by the corridors is modelled using 5 cubes and two cylinders which are unioned together. The corridors are then subtracted from a block. At this point the user wanted to position a skylight through the intersection of the corridors. Unsure of the exact positioning required, or the sort of results possible, the user roughly positioned a cylinder (the hole) and subtracted it from the model. A transparent instance of the primitive is also displayed by the system for reference. A manipulator was then used to drag the hole through the model revealing an unexpected new form. When satisfied with the positioning the hole is "fixed" in position. The fixing process doesn't change the internal representation of the model (it's still a complete CSG tree). It merely hides the apparatus used for interactive manipulation of the hole. The hole can be unfixed at any time and repositioned. This process of rough positioning, boolean combination and precise editing is then repeated.

6. Performance

The time complexity of our algorithm is proportional to the number of rendering operations carried out. We shall consider the rendering of one surface as a single rendering operation. Each pixel oriented "bookkeeping" operation is considered as an equivalent single unit. These operations have a lower geometry overhead than surface rendering but access more pixels. Equivalent functionality could be achieved by performing the bookkeeping operations with a repeated surface render. As in [?], we ignore the negligible normalization and pruning cost. We present the results for our current implementation of the algorithm. For reasons of clarity, some operations are described separately in section ??, whilst being implemented as a single operation.

Table ?? shows the number of rendering operations required for simple steps within the algorithm. The rendering algorithm is $O((kj)^2)$ for each product where j is the number of primitives in the product and k is the convexity of the primitives. The number of products generated by tree normalization is dependent on the structure of the tree and the geometry of the primitives with a worst case exponential relationship between number of primitives and products. In practice, both we, and Goldfeather et. al. [?], have found that the number of products after pruning is between $O(n)$ and $O(n^2)$ in the total number of primitives. The average product length, j, tends to be small and independent of the total number of primitives. Where long products arise they tend to be of the form $A-B-C-D-E$... and are susceptible to difference pruning.

Table ?? provides performance statistics for the eight sample models in figure ??. The images are 500 by 500 pixels and were rendered on a Silicon Graphics 5 span 310/VGXT with a single 33Mhz R3000 processor. The VGXT has an 8 bit stencil buffer. The first part of the table provides statistics on normalization and pruning. We include the number of primitives in the CSG expression, total triangles used to represent the primitives and the number of passes required. The number and average length of partial products produced by normalization with and without pruning are given. The second part of the table provides a breakdown of rendering operations into target rendering, classification & trimming and bookkeeping operations. The third part of the table provides a breakdown of rendering time in seconds; both for rendering operations and depth buffer save/restore time. The depth buffer save/restore time is given for the general case algorithm and for the optimization possible when the model is the first thing rendered in the current frame.

Table ?? shows rendering times together with number of passes required for different stencil buffer sizes. The increases in time are modest because the implementation only saves and restores the areas of the

Convexity	1-convex ($k = 1$)		k-convex	
Clipping	None	Near	None	Near
Classify Target Surface	k	k	$k+1$	$k+1$
Trimming Primitive	$2k+1$	$4k+1$	$2k+1$	$4k+1$
Render Target Surface	k	k	$k+1$	$k+1$

Table 3: *Rendering Operations per Step*

Model	a	b	c	d	e	f	g	h(part)	h(full)
Primitives	2	4	7	31	4	8	2	12	72
Triangles	96	256	408	1532	176	496	1928	8888	5536
Partial Products	2	6	32	34	5	14	3	13	72
Average Length	2	3	4	20.4	2.6	7	2	1.2	2.6
Partial Products (pruned)	2	6	32	34	5	14	3	13	72
Average Length (pruned)	2	3	4	2.7	2.6	3	2	1.2	2.3
Passes	1	1	1	5	1	2	1	1	11
Target Render Ops	2	4	7	30	4	8	5	25	72
Classification & Trimming Ops	4	18	128	92	13	42	8	8	164
Bookkeeping Render Ops	4	18	128	92	13	42	10	10	164
Total Render Ops	10	40	263	214	30	92	23	43	400
Target Time	0.005	0.003	0.038	0.039	0.008	0.017	0.031	0.009	0.118
Classification & Trimming Time	0.026	0.049	0.405	0.268	0.052	0.104	0.033	0.011	0.178
Bookkeeping Time	0.023	0.056	0.180	0.197	0.024	0.108	0.016	0.078	0.098
Save and Restore Time (general)	0.103	0.100	0.114	0.239	0.088	0.217	0.039	0.009	0.236
Save and Restore Time (first)	0.004	0.004	0.001	0.136	0.001	0.111	0.002	0.000	0.214
Total Time (general)	0.165	0.215	0.668	0.772	0.182	0.434	0.126	0.075	0.673
Total Time (first)	0.066	0.119	0.555	0.669	0.095	0.328	0.089	0.067	0.650

Table 4: *Rendering times (seconds) and statistics*

Stencil Bits	8	7	6	5	4	3	2
Model (c)	0.668(1)	0.670(2)	0.701(2)	0.735(2)	0.763(3)	0.782(4)	0.801(7)
Model (d)	0.772(5)	0.779(5)	0.804(6)	0.818(8)	0.837(10)	0.866(15)	0.884(30)
Model (f)	0.434(2)	0.435(2)	0.442(2)	0.426(2)	0.449(3)	0.475(4)	0.504(8)

Table 5: *Rendering time and number of passes with varying stencil size*

depth buffer that are changed during the classification stage. If less work is done in each pass the changed depth buffer areas typically become smaller. There is scope for further optimization of save and restore as the variations in times for the same number of passes shows. The different stencil buffer size causes a change in the composition of product groups. Placing partial products whose projected bounding boxes overlap into the same product groups will reduce the total area to be saved and restored.

Our algorithm performs particularly well in the sort of situations encountered within our interactive modelling system. Typically there is only ever one "dynamically" rendered CSG expression, usually involving a simple 1-convex "tool" and a more complex "workpiece" (figure ??(g)). Often we can achieve better performance by ignoring the top most caches of complex workpieces in order to expose more of the CSG tree to pruning. For example, in figure ??(h) an expression like $(A \cup B \cup C \cup D \cup ..) - X$ can be pruned to $A - X \cup B \cup C \cup D \cup ...$ This can vastly reduce both the number of polygons to be rendered (about 3–5 times as many polygons have to be rendered for A–B compared to $A \cup B$) and the size of the screen area involved in bookkeeping and depth buffer save and restore operations. We provide rendering times for both cached (table ?? h(full)) and uncached cases (table ?? h(part)) of figure ??(h). The coloured primitives are those that are being "moved", the other geometry can be rendered from caches. The version that makes use of the caches is about 9 times faster than the fully rendered version. However, the triangle count is higher because the cached geometry has a more complex boundary than the original primitives.

Our implementation's performance compares well with that obtained by specialized hardware and pure software solutions. Figure ??(d) is our version of a model rendered by Goldfeather et. al. [?] on Pixel-Planes 4. They report a total rendering time of 4.02 seconds compared with our time of 0.67 seconds. The VGX architecture machine used for our tests was introduced in 1990 when Pixel-Planes 4 was nearing the end of its lifetime. Pixel-Planes 5 (the most recent machine in the Pixel-Planes series [?]) has performance some 50 times better than Pixel-Planes 4 on a full system with 32 geometry processors and 16 renderers. Such a system would have performance 10 times that of our implementation — at a far greater cost.

Figure ??(f) is our version of a model rendered by Thibault and Naylor's BSP tree based algorithm [?]. Their total rendering time is 7.2 seconds for a model with 158 polygons on a VAX 8650. Our time is 0.3 seconds for a model with 496 triangles. Our algorithm also scales better with increasing numbers of polygons (*O(kn)* compared with *O(nlogn)*).

6.1. Other implementations

We have also implemented the algorithm using OpenGL [?] and tested it on our VGXT, a Silicon Graphics R3000 Indigo with starter graphics, and an Indigo^2 Extreme. The algorithm should run under any OpenGL implementation. On the systems we tested performance was comparable to the GL version in all areas except depth buffer save and restore. This operation was about 100 times slower than the GL equivalent. The problem appears to be a combination of poor performance tuning and a specification which requires conversion of the depth buffer values to and from normalized floating point. This problem should be resolved with the release of more mature OpenGL implementations. Single pass renders with the frame start optimization (the common case for our interactive modeller) run at full speed.

7. Conclusion

We have presented an algorithm which directly renders an arbitrary CSG tree and is suitable for use in interactive modelling applications. Unlike Goldfeather et. al. [?], our algorithm requires only a single color buffer, a single depth buffer, a stencil buffer and the ability to save and restore the contents of the depth buffer. It can be implemented on many graphics workstations using existing graphics libraries. Like Rossignac, Megahed and Schneider [?], the algorithm can display cross-sections of solids using clipping planes but is far more flexible. For instance, the algorithm could be used to directly display interferences between solids by rendering the intersection of the solids.

The algorithm has been implemented on an SGI 310/VGXT using the GL graphics library and has been integrated into an experimental modelling system. Performance compares well with specialized hardware and pure software algorithms for complete evaluation and rendering. The algorithm performs particularly well for incremental updates in an interactive modelling environment.

Acknowledgements

This work has been funded by Informatix Inc., Tokyo. Our thanks go to them for their support of the Martin Centre CADLAB over the last four years. Brian Logan, Paul Richens and Simon Schofield have all provided valuable insights and comments; as have the anonymous referees. Paul Richens created the models in figure ?? (g) and (h) using our interactive modeller.

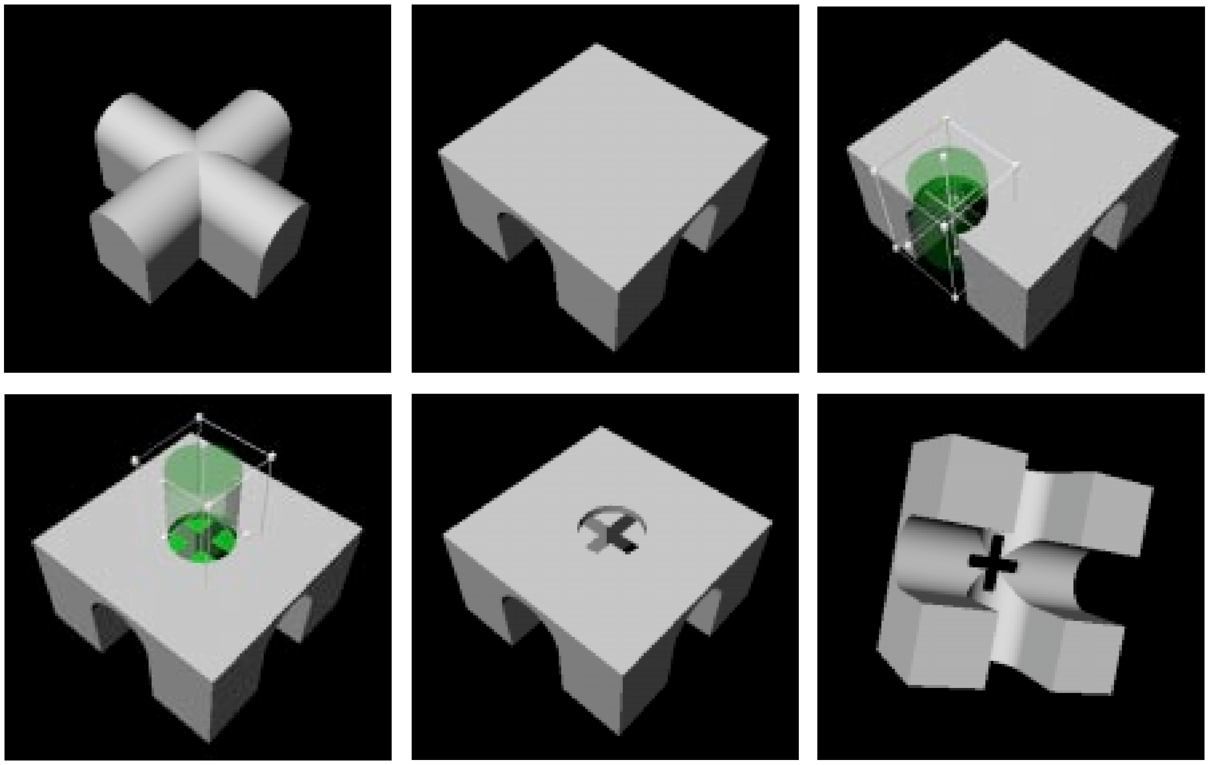
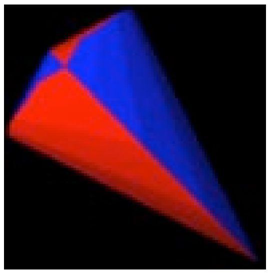
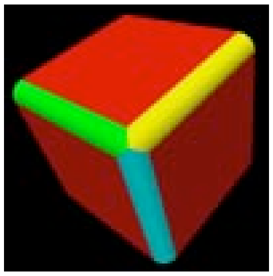


Figure 8: *Dragging a hole though the model reveals an unexpected new form*

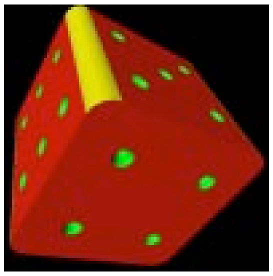


(a) A∩B

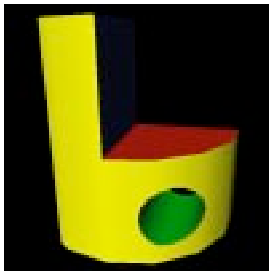
(b) (A∪B) (C∪D)



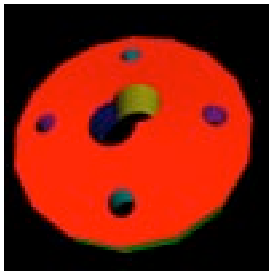
(c) ((A∩B)∪(A C))
∩((A∩D)∪(A E))
∩((A∩F)∪(A G))



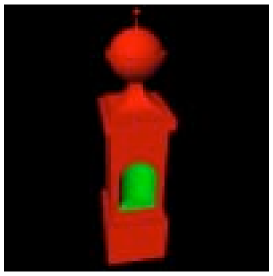
(d) (A∩B)∪(A∩C)∪
(A∩D)∪(A∩E)∪
(A F G H ...)



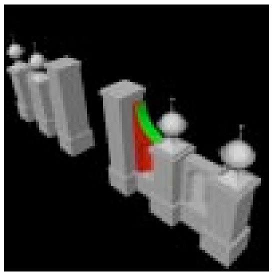
(e) (A∩D B)∪(C∩D)



(f) (A∪B) C ... H



(g) A_2 B



(h) A_2 X∪B∪C∪...

Figure 9: *Images generated by the stencil buffer CSG algorithm*

Appeared in Proc. Fourth Eurographics Workshop on Rendering,
Michael Cohen, Claude Puech, and Francois Sillion, eds.
Paris, France, June, 1993. pp. 259-266.

Texture Mapping as a Fundamental Drawing Primitive

Paul Haeberli
Mark Segal
Silicon Graphics Computer Systems*

Abstract

Texture mapping has traditionally been used to add realism to computer graphics images. In recent years, this technique has moved from the domain of software rendering systems to that of high performance graphics hardware.

But texture mapping hardware can be used for many more applications than simply applying diffuse patterns to polygons.

We survey applications of texture mapping including simple texture mapping, projective textures, and image warping. We then describe texture mapping techniques for drawing anti-aliased lines, air-brushes, and anti-aliased text. Next we show how texture mapping may be used as a fundamental graphics primitive for volume rendering, environment mapping, color interpolation, contouring, and many other applications.

CR Categories and Subject Descriptors: I.3.3 **[Computer Graphics]:** Picture/Image Generation; I.3.7 **[Computer Graphics]:** Three-Dimensional Graphics and Realism - *color, shading, shadowing, texture-mapping, line drawing, and anti-aliasing*

1 Introduction

Texture mapping[Cat74][Hec86] is a powerful technique for adding realism to a computer-generated scene. In its basic form, texture mapping lays an image (the texture) onto an object in a scene. More general forms of texture mapping generalize the image to other information; an "image" of altitudes, for instance, can be used to control shading across a surface to achieve such effects as bump-mapping.

Because texture mapping is so useful, it is being provided as a standard rendering technique both in graphics software interfaces and in computer graphics hardware[HL90][DWS$^+$88]. Texture mapping can therefore be used in a scene with only a modest increase in the complexity of the program that generates that scene, sometimes with little effect on scene generation time. The wide availability and high-performance of texture mapping makes it a desirable rendering technique for achieving a number of effects that are normally obtained with special purpose drawing hardware.

After a brief review of the mechanics of texture mapping, we describe a few of its standard applications. We go on to describe some novel applications of texture mapping.

2 Texture Mapping

When mapping an image onto an object, the color of the object at each pixel is modified by a corresponding color from the image. In general, obtaining this color from the image conceptually requires several steps[Hec89]. The image is normally stored as a sampled array, so a continuous image must first be reconstructed from the samples. Next, the image must be warped to match any distortion (caused, perhaps, by perspective) in the projected object being displayed. Then this warped image is filtered to remove high-frequency components that would lead to aliasing in the final step: resampling to obtain the desired color to apply to the pixel being textured.

In practice, the required filtering is approximated by one of several methods. One of the most popular is *mipmapping*[Wil83]. Other filtering techniques may also be used[Cro84].

There are a number of generalizations to this basic texture mapping scheme. The image to be mapped need not be two-dimensional; the sampling and filtering techniques may be applied for both one- and three-dimensional images[Pea85]. In the case of a three-dimensional image, a two-dimensional slice must be selected to be mapped onto an object's boundary, since the result of rendering must be two-dimensional. The

*2011 N. Shoreline Blvd., Mountain View, CA 94043 USA image may not be stored as an array but may be procedurally generated[Pea85][Per85]. Finally, the image may not represent color at all, but may instead describe transparency or other surface properties to be used in lighting or shading calculations[CG85].

3 Previous Uses of Texture Mapping

In basic texture mapping, an image is applied to a polygon (or some other surface facet) by assigning texture coordinates to the polygon's vertices. These coordinates index a texture image, and are interpolated across the polygon to determine, at each of the polygon's pixels, a texture image value. The result is that some portion of the texture image is mapped onto the polygon when the polygon is viewed on the screen. Typical two-dimensional images in this application are images of bricks or a road surface (in this case the texture image is often repeated across a polygon); a three-dimensional image might represent a block of marble from which objects could be "sculpted."

3.1 Projective Textures

A generalization of this technique projects a texture onto surfaces as if the texture were a projected slide or movie[SKvW$^+$92]. In this case the texture coordinates at a vertex are computed as the result of the projection rather than being assigned fixed values. This technique may be used to simulate spotlights as well as the reprojection of a photograph of an object back onto that object's geometry.

Projective textures are also useful for simulating shadows. In this case, an image is constructed that represents distances from a light source to surface points nearest the light source. This image can be computed by performing z-buffering from the light's point of view and then obtaining the resulting z-buffer. When the scene is viewed from the eyepoint, the distance from the light source to each point on a surface is computed and compared to the corresponding value stored in the texture image. If the values are (nearly) equal, then the point is not in shadow; otherwise, it is in shadow. This technique should not use mipmapping, because filtering must be applied *after* the shadow comparison is performed[RSC87].

3.2 Image Warping

Image warping may be implemented with texture mapping by defining a correspondence between a uniform polygonal mesh (representing the original image) and a warped mesh (representing the warped image)[OTOK87]. The warp may be affine (to generate rotations, translations, shearings, and zooms) or higher-order. The points of the warped mesh are assigned the corresponding texture coordinates of the uniform mesh, and the mesh is texture mapped with the original image. This technique allows for easily-controlled interactive image warping. The technique can also be used for panning across a large texture image by using a mesh that indexes only a portion of the entire image.

3.3 Transparency Mapping

Texture mapping may be used to lay transparent or semi-transparent objects over a scene by representing transparency values in the texture image as well as color values. This technique is useful for simulating clouds[Gar85] and trees for example, by drawing appropriately textured polygons over a background. The effect is that the background shows through around the edges of the clouds or branches of the trees. Texture map filtering applied to the transparency and color values automatically leads to soft boundaries between the clouds or trees and the background.

3.4 Surface Trimming

Finally, a similar technique may be used to cut holes out of polygons or perform domain space trimming on curved surfaces[Bur92]. An image of the domain space trim regions is generated. As the surface is rendered, its domain space coordinates are used to reference this image. The value stored in the image determines whether the corresponding point on the surface is trimmed or not.

4 Additional Texture Mapping Applications

Texture mapping may be used to render objects that are usually rendered by other, specialized means. Since it is becoming widely available, texture mapping may be a good choice to implement these techniques even when these graphics primitives can be drawn using special purpose methods.

4.1 Anti-aliased Points and Line Segments

One simple use of texture mapping is to draw anti-aliased points of any width. In this case the texture image is of a filled circle with a smooth (anti-aliased) boundary. When a point is specified, it's coordinates indicate the center of a square whose width is determined by the point size. The texture coordinates at the

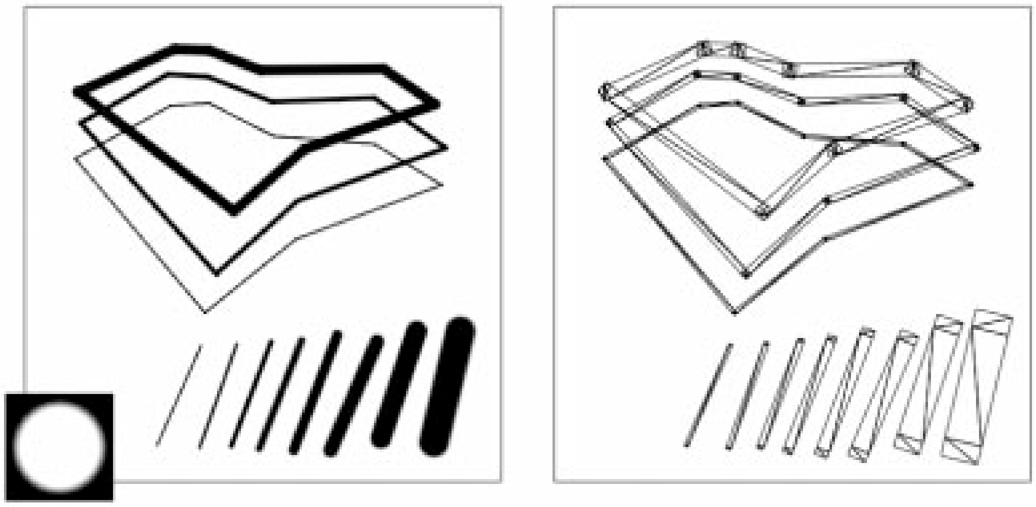


Figure 1. Anti-aliased line segments.

square's corners are those corresponding to the corners of the texture image. This method has the advantage that any point shape may be accommodated simply by varying the texture image.

A similar technique can be used to draw anti-aliased, line segments of any width[Gro90]. The texture image is a filtered circle as used above. Instead of a line segment, a texture mapped rectangle, whose width is the desired line width, is drawn centered on and aligned with the line segment. If line segments with round ends are desired, these can be added by drawing an additional textured rectangle on each end of the line segment (Figure 1).

4.2 Air-brushes

Repeatedly drawing a translucent image on a background can give the effect of spraying paint onto a canvas. Drawing an image can be accomplished by drawing a texture mapped polygon. Any conceivable brush "footprint", even a multi-colored one, may be drawn using an appropriate texture image with red, green, blue, and alpha. The brush image may also easily be scaled and rotated (Figure 2).

4.3 Anti-aliased Text

If the texture image is an image of a character, then a polygon textured with that image will show that character on its face. If the texture image is partitioned into an array of rectangles, each of which contains the image of a different character, then any character may be displayed by drawing a polygon with appropriate texture coordinates assigned to its vertices. An advantage of this method is that strings of characters may be arbitrarily positioned and oriented in three dimensions by appropriately positioning and orienting the textured polygons. Character kerning is accomplished simply by positioning the polygons relative to one another (Figure 3).

Antialiased characters of any size may be obtained with a single texture map simply by drawing a polygon of the desired size, but care must be taken if mipmapping is used. Normally, the smallest mipmap is 1 pixel square, so if all the characters are stored in a single texture map, the smaller mipmaps will contain a number of characters filtered together. This will generate undesirable effects when displayed characters are too small. Thus, if a single texture image is used for all characters, then each must be carefully placed in the image, and mipmaps must stop at the point where the image of a single character is reduced to 1 pixel on a side. Alternatively, each character could be placed in its own (small) texture map.

4.4 Volume Rendering

There are three ways in which texture mapping may be used to obtain an image of a solid, translucent object. The first is to draw slices of the object from back to front[DCH88]. Each slice is drawn by first generating a texture image of the slice by sampling the data representing the volume along the plane of the slice, and then drawing a texture mapped polygon to produce the slice. Each slice is blended with the previously drawn slices using transparency.

The second method uses 3D texture mapping[Dre92]. In this method, the volumetric data is copied into the 3D texture image. Then, slices perpendicular to the viewer are drawn. Each slice is again a texture mapped

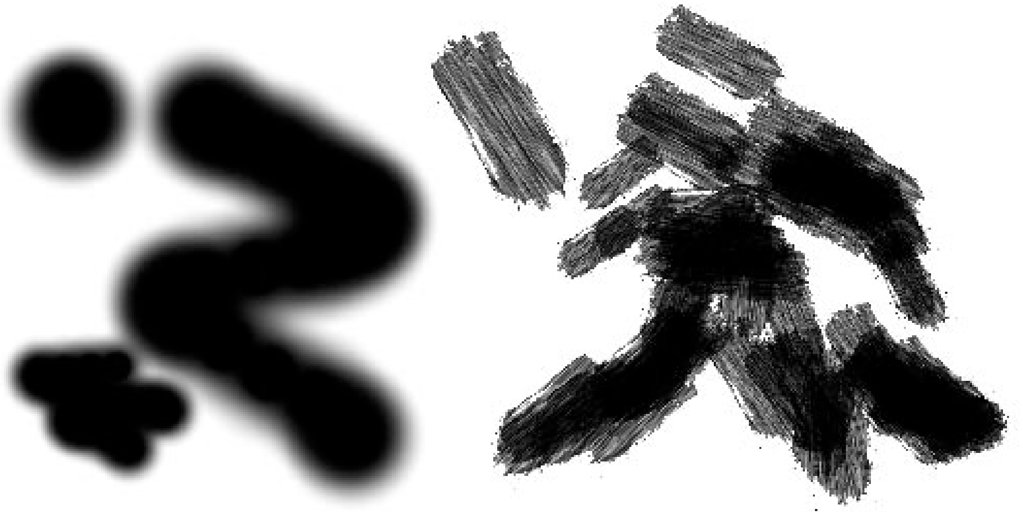


Figure 2. Painting with texture maps.

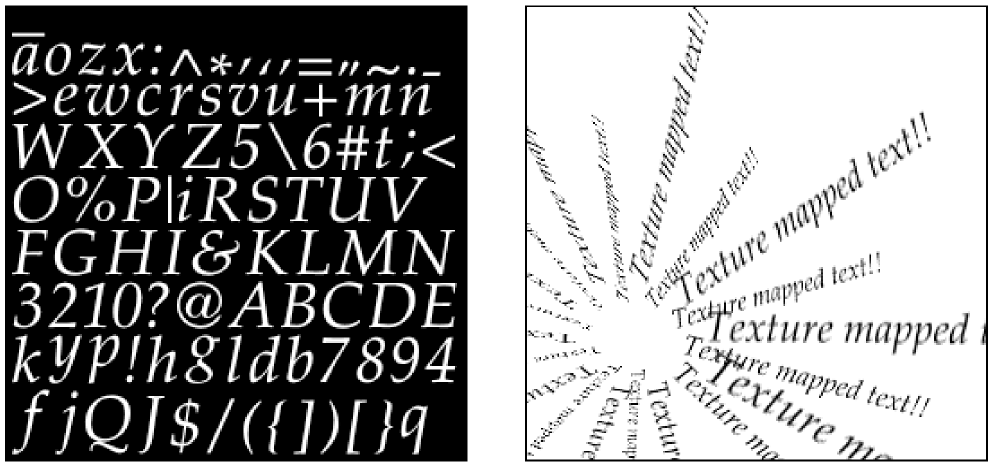


Figure 3. Anti-aliased text.

polygon, but this time the texture coordinates at the polygon's vertices determine a slice through the 3D texture image. This method requires a 3D texture mapping capability, but has the advantage that texture memory need be loaded only once no matter what the viewpoint. If the data are too numerous to fit in a single 3D image, the full volume may be rendered in multiple passes, placing only a portion of the volume data into the texture image on each pass.

A third way is to use texture mapping to implement "splatting" as described by[Wes90][LH91].

4.5 Movie Display

Three-dimensional texture images may also be used to display animated sequences[Ake92]. Each frame forms one two-dimensional slice of a three-dimensional texture. A frame is displayed by drawing a polygon with texture coordinates that select the desired slice. This can be used to smoothly interpolate between frames of the stored animation. Alpha values may also be associated with each pixel to make animated "sprites".

4.6 Contouring

Contour curves drawn on an object can provide valuable information about the object's geometry. Such curves may represent height above some plane (as in a topographic map) that is either fixed or moves with the object[Sab88]. Alternatively, the curves may indicate intrinsic surface properties, such as geodesics or loci of constant curvature.

Contouring is achieved with texture mapping by first defining a one-dimensional texture image that is of con-

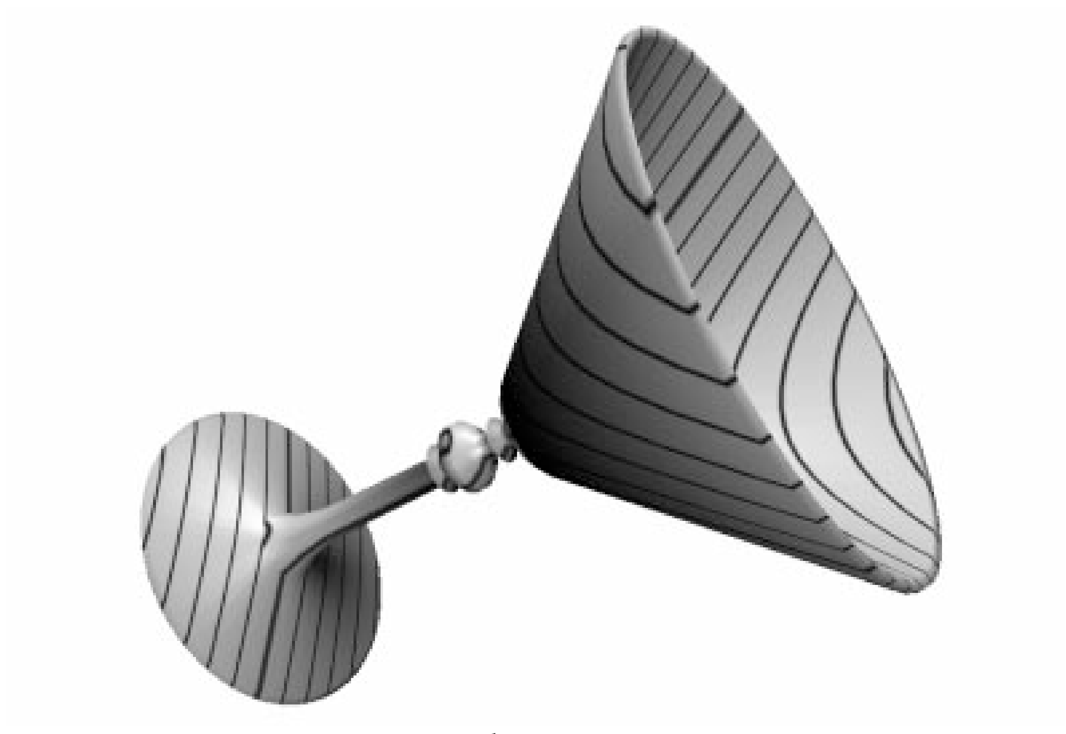


Figure 4. Contouring showing distance from a plane.

stant color except at some spot along its length. Then, texture coordinates are computed for vertices of each polygon in the object to be contoured using a *texture coordinate generation function*. This function may calculate the distance of the vertex above some plane (Figure 4), or may depend on certain surface properties to produce, for instance, a curvature value. Modular arithmetic is used in texture coordinate interpolation to effectively cause the single linear texture image to repeat over and over. The result is lines across the polygons that comprise an object, leading to contour curves.

A two-dimensional (or even three-dimensional) texture image may be used with two (or three) texture coordinate generation functions to produce multiple curves, each representing a different surface characteristic.

4.7 Generalized Projections

Texture mapping may be used to produce a non-standard projection of a three-dimensional scene, such as a cylindrical or spherical projection[Gre86]. The technique is similar to image warping. First, the scene is rendered six times from a single viewpoint, but with six distinct viewing directions: forward, backward, up, down, left, and right. These six views form a cube enclosing the viewpoint. The desired projection is formed by projecting the cube of images onto an array of polygons (Figure 5).

4.8 Color Interpolation in non-RGB Spaces

The texture image may not represent an image at all, but may instead be thought of as a lookup table. Intermediate values not represented in the table are obtained through linear interpolation, a feature normally provided to handle image filtering.

One way to use a three-dimensional lookup table is to fill it with RGB values that correspond to, for instance, HSV (Hue, Saturation, Value) values. The H, S, and V values index the three dimensional tables. By assigning HSV values to the vertices of a polygon linear color interpolation may be carried out in HSV space rather than RGB space. Other color spaces are easily supported.

4.9 Phong Shading

Phong shading with an infinite light and a local viewer may be simulated using a 3D texture image as follows. First, consider the function of x, y, and z that assigns a brightness value to coordinates that represent a (not necessarily unit length) vector. The vector is the reflection off of the surface of the vector from the eye to a point on the surface, and is thus a function of the normal at that point. The brightness function depends on the location of the light source. The 3D texture image is a lookup table for the brightness function given a reflection vector. Then, for each polygon in the scene, the reflection vector is computed at each of the polygon's vertices. The coordinates of this vector are interpolated

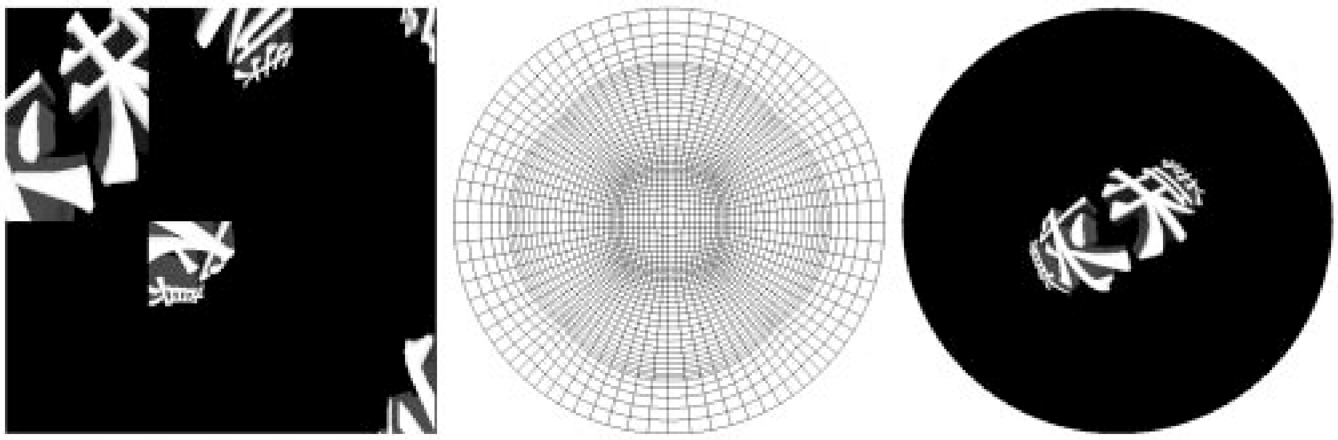


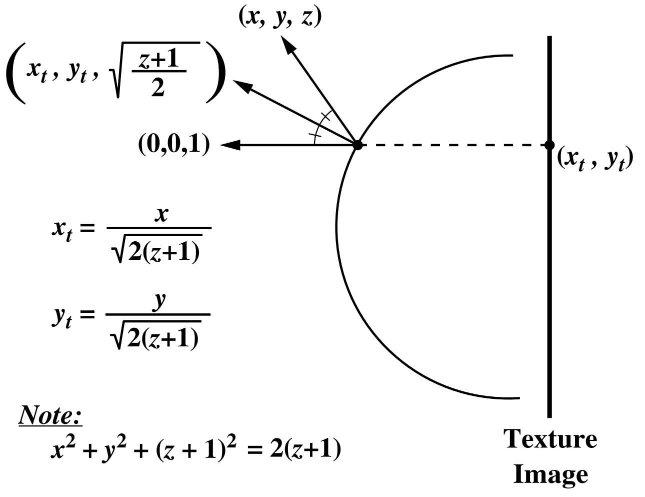
Figure 5. 360 Degree fisheye projection.

across the polygon and index the brightness function stored in the texture image. The brightness value so obtained modulates the color of the polygon. Multiple lights may be obtained by incorporating multiple brightness functions into the texture image.

4.10 Environment Mapping

Environment mapping[Gre86] may be achieved through texture mapping in one of two ways. The first way requires six texture images, each corresponding to a face of a cube, that represent the surrounding environment. At each vertex of a polygon to be environment mapped, a reflection vector from the eye off of the surface is computed. This reflection vector indexes one of the six texture images. As long as all the vertices of the polygon generate reflections into the same image, the image is mapped onto the polygon using projective texturing. If a polygon has reflections into more than one face of the cube, then the polygon is subdivided into pieces, each of which generates reflections into only one face. Because a reflection vector is not computed at each pixel, this method is not exact, but the results are quite convincing when the polygons are small.

The second method is to generate a single texture image of a perfectly reflecting sphere in the environment. This image consists of a circle representing the hemisphere of the environment behind the viewer, surrounded by an annulus representing the hemisphere in front of the viewer. The image is that of a perfectly reflecting sphere located in the environment when the viewer is infinitely far from the sphere. At each polygon vertex, a texture coordinate generation function generates coordinates that index this texture image, and these are interpolated across the polygon. If the (normalized) reflection vector at a vertex is $r = (\,x \quad y \quad z\,)$, and $m = \sqrt{2(z+1)}$, then the generated coordinates are x/m and y/m when the texture image is indexed


(x, y, z)
$\left(x_t\,, y_t\,, \sqrt{\frac{z+1}{2}}\right)$
(0,0,1)
$(x_t\,, y_t)$
$x_t = \frac{x}{\sqrt{2(z+1)}}$
$y_t = \frac{y}{\sqrt{2(z+1)}}$
Note:
$x^2 + y^2 + (z+1)^2 = 2(z+1)$
Texture Image
Figure 6. Spherical reflection geometry.

by coordinates ranging from -1 to 1. (The calculation is diagrammed in Figure 6). This method has the disadvantage that the texture image must be recomputed whenever the view direction changes, but requires only a single texture image with no special polygon subdivision (Figure 7).

4.11 3D Halftoning

Normal halftoned images are created by thresholding a source image with a halftone screen. Usually this halftone pattern of lines or dots bears no direct relationship to the geometry of the scene. Texture mapping allows halftone patterns to be generated using a 3D spatial function or parametric lines of a surface (Figure 8). This permits us to make halftone patterns that are bound to the surface geometry[ST90].

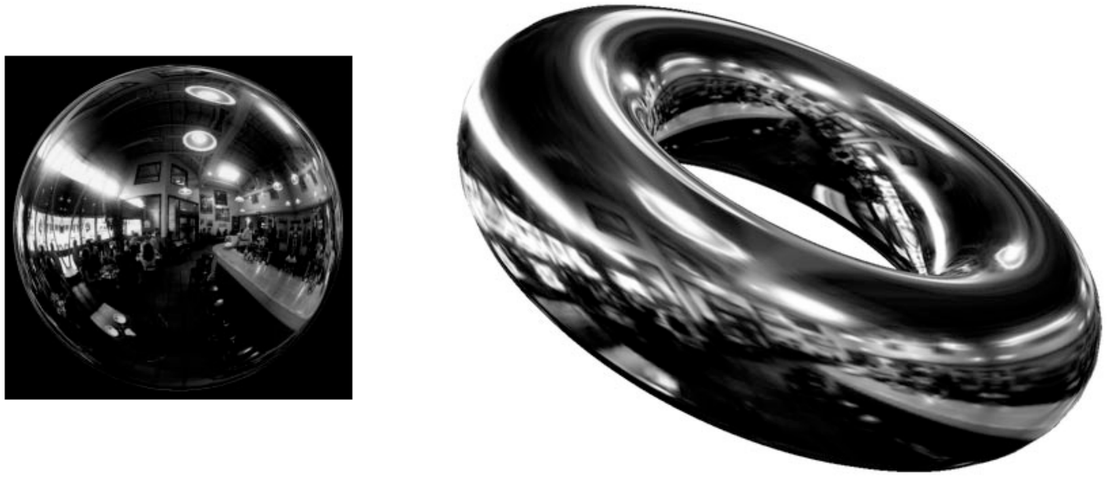


Figure 7. Environment mapping.

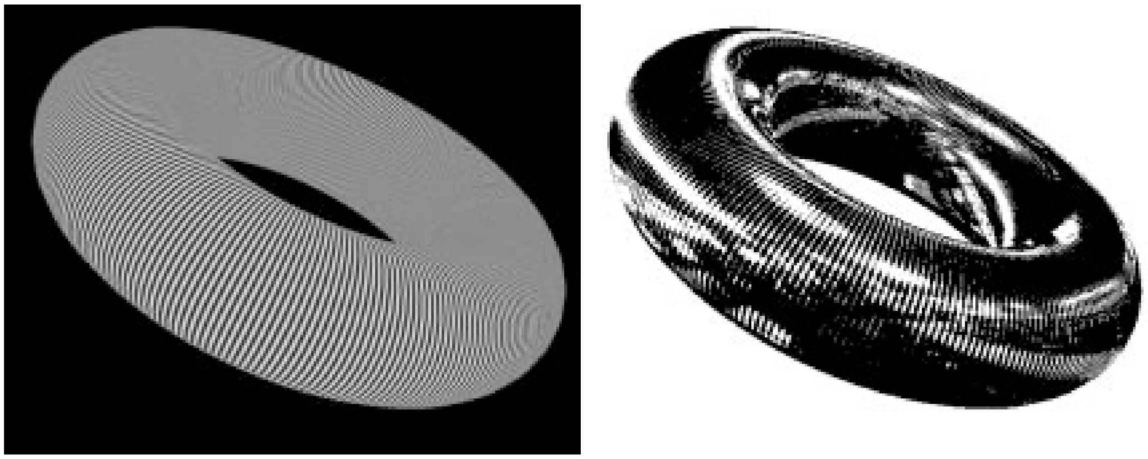


Figure 8. 3D halftoning.

5 Conclusion

Many graphics systems now provide hardware that supports texture mapping. As a result, generating a texture mapped scene need not take longer than generating a scene without texture mapping.

We have shown that, in addition to its standard uses, texture mapping can be used for a large number of interesting applications, and that texture mapping is a powerful and flexible low level graphics drawing primitive.

References

[Ake92] Kurt Akeley. Personal Communication, 1992.

[Bur92] Derrick Burns. Personal Communication, 1992.

[Cat74] Ed Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[CG85] Richard J. Carey and Donald P. Greenberg. Textures for realistic image synthesis. *Computers & Graphics*, 9(3):125–138, 1985.

[Cro84] F. C. Crow. Summed-area tables for texture mapping. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18:207–212, July 1984.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):65–74, August 1988.

[Dre92] Bob Drebin. Personal Communication, 1992.

[DWS+88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: A VLSI system for high performance graphics. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):21–30, August 1988.

[Gar85] G. Y. Gardner. Visual simulation of clouds. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):297–303, July 1985.

[Gre86] Ned Greene. Applications of world projections. *Proceedings of Graphics Interface '86*, pages 108–114, May 1986.

[Gro90] Mark Grossman. Personal Communication, 1990.

[Hec86] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, November 1986.

[Hec89] Paul S. Heckbert. Fundamentals of texture mapping and image warping. M.sc. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, June 1989.

[HL90] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):289–298, August 1990.

[LH91] David Laur and Pat Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):285–288, July 1991.

[OTOK87] Masaaki Oka, Kyoya Tsutsui, Akio Ohba, and Yoshitaka Kurauchi. Real-time manipulation of texture-mapped surfaces. *Computer Graphics (Proceedings of SIGGRAPH '87)*, July 1987.

[Pea85] D. R. Peachey. Solid texturing of complex surfaces. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):279–286, July 1985.

[Per85] K. Perlin. An image synthesizer. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):287–296, July 1985.

[RSC87] William Reeves, David Salesin, and Rob Cook. Rendering antialiased shadows with depth maps. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):283–291, July 1987.

[Sab88] Paolo Sabella. A rendering algorithm for visualizing 3d scalar fields. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):51–58, August 1988.

[SKvW+92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):249–252, July 1992.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):197–206, August 1990.

[Wes90] Lee Westover. Footprint evaluation for volume rendering. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):367–376, August 1990.

[Wil83] Lance Williams. Pyramidal parametrics. *Computer Graphics (SIGGRAPH '83 Proceedings)*, 17(3):1–11, July 1983.

Texture Mapping in Technical, Scientific and Engineering Visualization

Michael Teschner[1] *and Christian Henn*[2]

[1]*Chemistry and Health Industry Marketing,*
Silicon Graphics Basel, Switzerland

[2]*Maurice E. Mueller Institute for Microscopy,*
University of Basel, Switzerland

Executive Summary

As of today, texture mapping is used in visual simulation and computer animation to reduce geometric complexity while enhancing realism. In this report, this common usage of the technology is extended by presenting application models of real time texture mapping that solve a variety of visualization problems in the general technical and scientific world, opening new ways to represent and analyze large amounts of experimental or simulated data.

The topics covered in this report are:

- Abstract definition of the texture mapping concept
- Visualization of properties on surfaces by color coding
- Information filtering on surfaces
- Real time volume rendering concepts
- Quality enhanced surface rendering

In the following sections, each of these aspects will be described in detail. Implementation techniques are outlined using pseudo code that emphasizes the key aspects. A basic knowledge in GL programming is assumed. Application examples are taken from the chemical market. However, for the scope of this report no particular chemical background is required, since the data being analyzed can in fact be replaced by any other source of technical, scientific or engineering information processing.

Note, that this report discusses the potential of released advanced graphics technology in a very detailed fashion. The presented topics are based on recent and ongoing research and therefore subjected to change.

The methods described are the result of a team work involving scientists from different research areas and institutions, and is called the *Texture Team,* consisting of the following members:

- Prof. Juergen Brickmann, Technische Hochschule, Darmstadt, Germany
- Dr. Peter Fluekiger, Swiss Scientific Computing Center, Manno, Switzerland
- Christian Henn, M.E. Mueller Institute for Microscopy, Basel, Switzerland
- Dr. Michael Teschner, Silicon Graphics Marketing, Basel, Switzerland

Further support came from SGI's Advanced Graphics Division engineering group.

Colored pictures and sample code are available from sgigate.sgi.com via anonymous ftp. The files will be there starting November 1st 1993 and will be located in the directory pub/SciTex.

For more information, please contact:

Michael Teschner	*(41) 61 67 09 03*	*(phone)*
SGI Marketing, Basel	*(41) 61 67 12 01*	*(fax)*
Erlenstraesschen 65		
CH 4125 Riehen, Switzerland	*micha@basel.sgi.com*	*(email)*

1 Introduction

Texture mapping [1,2] has traditionally been used to add realism in computer generated images. In recent years, this technique has been transferred from the domain of software based rendering systems to a hardware supported feature of advanced graphics workstations. This was largely motivated by visual simulation and computer animation applications that use texture mapping to map pictures of surface texture to polygons of 3 D objects [3].

Thus, texture mapping is a very powerful approach to add a dramatic amount of realism to a computer generated image without blowing up the geometric complexity of the rendered scenario, which is essential in visual simulators that need to maintain a constant frame rate. E.g., a realistically looking house can be displayed using only a few polygons with photographic pictures of a wall showing doors and windows being mapped to. Similarly, the visual richness and accuracy of natural materials such as a block of wood can be improved by wrapping a wood grain pattern around a rectangular solid.

Up to now, texture mapping has not been used in technical or scientific visualization, because the above mentioned visual simulation methods as well as non interactive rendering applications like computer animation have created a severe bias towards what texture mapping can be used for, i.e. wooden [4] or marble surfaces for the display of solid materials, or fuzzy, stochastic patterns mapped on quadrics to visualize clouds [5,6].

It will be demonstrated that hardware supported texture mapping can be applied in a much broader range of application areas. Upon reverting to a strict and formal definition of texture mapping that generalizes the texture to be a general repository for pixel based color information being mapped on arbitrary 3 D geometry, a powerful and elegant framework for the display and analysis of technical and scientific information is obtained.

2 Abstract definition of the texture mapping concept

In the current hardware implementation of SGI [7], texture mapping is an additional capability to modify pixel information during the rendering procedure, after the shading operations have been completed. Although it modifies pixels, its application programmers interface is vertex based. Therefore texture mapping results in only a modest or small increase in program complexity. Its effect on the image generation time depends on the particular hardware being used: entry level and interactive systems show a significant performance reduction, whereas on third generation graphics subsystems texture mapping may be used without any performance penalty.
Three basic components are needed for the texture mapping procedure: (1) the texture, which is defined in the texture space, (2) the 3 D geometry, defined on a per vertex basis and (3) a mapping function that links the texture to the vertex description of the 3 D object.

The texture space [8,9] is a parametric coordinate space which can be 1,2 or 3 dimensional. Analogous to the pixel (picture element) in screen space, each element in texture space is called texel (texture element). Current hardware implementations offer flexibility with respect to how the information stored with each texel is interpreted. Multi channel colors, intensity, transparency or even lookup indices corresponding to a color lookup table are supported.

In an abstract definition of texture mapping, the texture space is far more than just a picture within a parametric coordinate system: the texture space may be seen as a special memory segment, where a variety of information can be deposited which is then linked to object representations in 3 D space. Thus this information can efficiently be used to represent any parametric property that needs to be visualized.

Although the vertex based nature of 3 D geometry in general allows primitives such as points or lines to be texture mapped as well, the real value of texture mapping emerges upon drawing filled triangles or higher order polygons.

The mapping procedure assigns a coordinate in texture space to each vertex of the 3 D object. It is important to note that the dimensionality of the texture space is independent from the dimensionality of the displayed object. E.g., coding a simple property into a 1 D texture can be used to generate isocontour lines on arbitrary 3 D surfaces.

3 Color coding based application solutions

Color coding is a popular means of displaying scalar information on a surface [10]. E.g., this can be used to display stress on mechanical parts or interaction potentials on molecular surfaces.

The problem with traditional, Gouraud shading based implementations occurs when there is a high contrast color code variation on sparsely tesselated geometry: since the color coding is done by assigning RGB color triplets to the vertices of the 3 D geometry, pixel colors will be generated by linear interpolation in RGB color space.

As a consequence, all entries in the defined color ramp laying outside the linear color ramp joining two RGB triplets are never taken into account and information will be lost. In Figure 1, a symmetric grey scale covering the property range is used to define the color ramp. On the left hand side, the interpolation in the RGB color space does not reflect the color ramp. There is a substantial loss of information during the rendering step.

With a highly tessellated surface, this problem can be reduced. An alignment of the surface vertices with the expected color code change or multi pass rendering may remove such artifacts completely. However, these methods demand large numbers of polygons or extreme algorithmic complexity, and are therefore not suited for interactive applications.

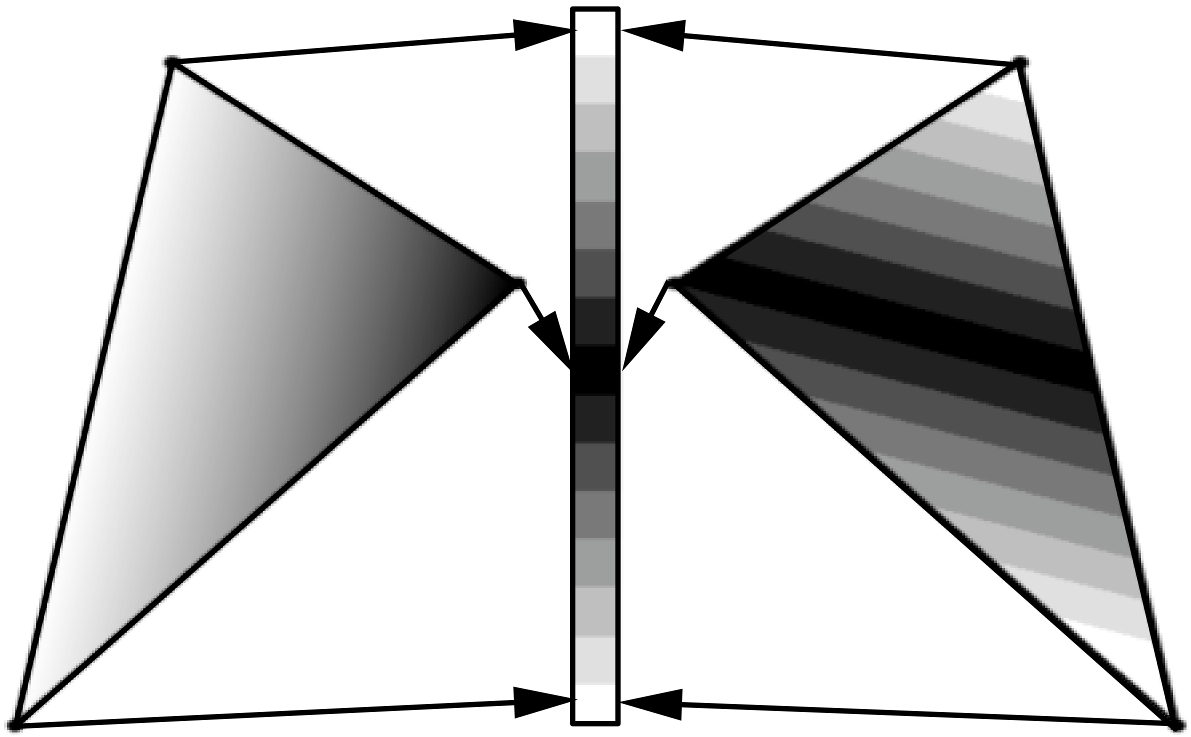


Figure 1: *Color coding with RGB interpolation (left) and texture mapping (right).*

This problem can be solved by storing the color ramp as a 1 D texture. In contrast to the above described procedure, the scalar property information is used as the texture coordinates for the surface vertices. The color interpolation is then performed in the texture space, i.e. the coloring is evaluated at every pixel (Figure 1 right). High contrast variation in the color code is now possible, even on sparsely tessellated surfaces.

It is important to note that, although the texture is one dimensional, it is possible to tackle a 3 D problem. The dimensionality of texture space and object space is independent, thus they do not affect each other. This feature of the texture mapping method, as well as the difference between texture interpolation and color interpolation is crucial for an understanding of the applications presented in this report.

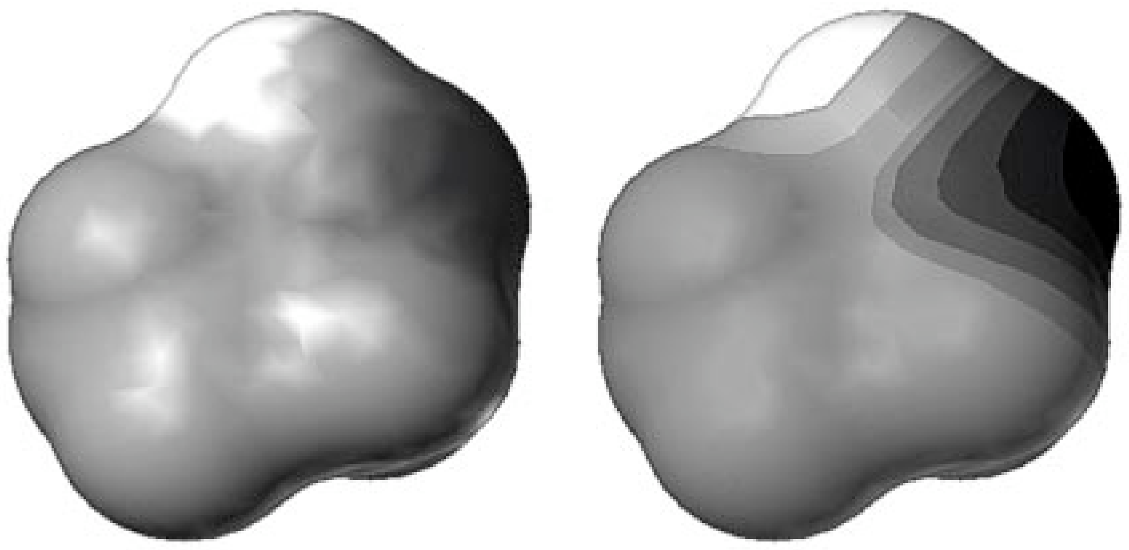


Figure 2: *Electrostatic potential coded on the solvent accessible surface of ethanol.*

Figure 2 shows the difference between the two procedures with a concrete example: the solvent accessible surface of the ethanol molecule is colored by the electrostatic surface potential, using traditional RGB color interpolation (left) and texture mapping (right).

The independence of texture and object coordinate space has further advantages and is well suited to accommodate immediate changes to the meaning of the color ramp. E.g., by applying a simple 3 D transformation like a translation in texture space the zero line of the color code may be shifted. Applying a scaling transformation to the texture adjusts the range of the mapping. Such modifications may be performed in real time.

With texture mapping, the resulting sharp transitions from one color value to the next significantly improves the rendering accuracy. Additionally, these sharp transitions help to visually understand the object's 3 D shape.

3.1 Isocontouring on surfaces

Similar to the color bands in general color coding, discrete contour lines drawn on an object provide valuable information about the object's geometry as well as its properties, and are widely used in visual analysis applications. E.g., in a topographic map they might represent height above some plane that is either fixed in world coordinates or moves with the object [11]. Alternatively, the curves may indicate intrinsic surface properties, such as an interaction potential or stress distributions.

With texture mapping, discrete contouring may be achieved using the same setup as for general color coding. Again, the texture is 1 D, filled with a base color that represents the objects surface appearance. At each location of a contour threshold, a pixel is set to the color of the particular threshold. Figure 3 shows an application of this texture to display the hydrophobic potential of Gramicidine A, a channel forming molecule as a set of isocontour lines on the surface of the molecular surface.

Scaling of the texture space is used to control the spacing of contour thresholds. In a similar fashion, translation of the texture space will result in a shift of all threshold values. Note that neither the underlying geometry nor the texture itself was modified during this procedure. Adjustment of the threshold spacing is performed in real time, and thus fully interactive.

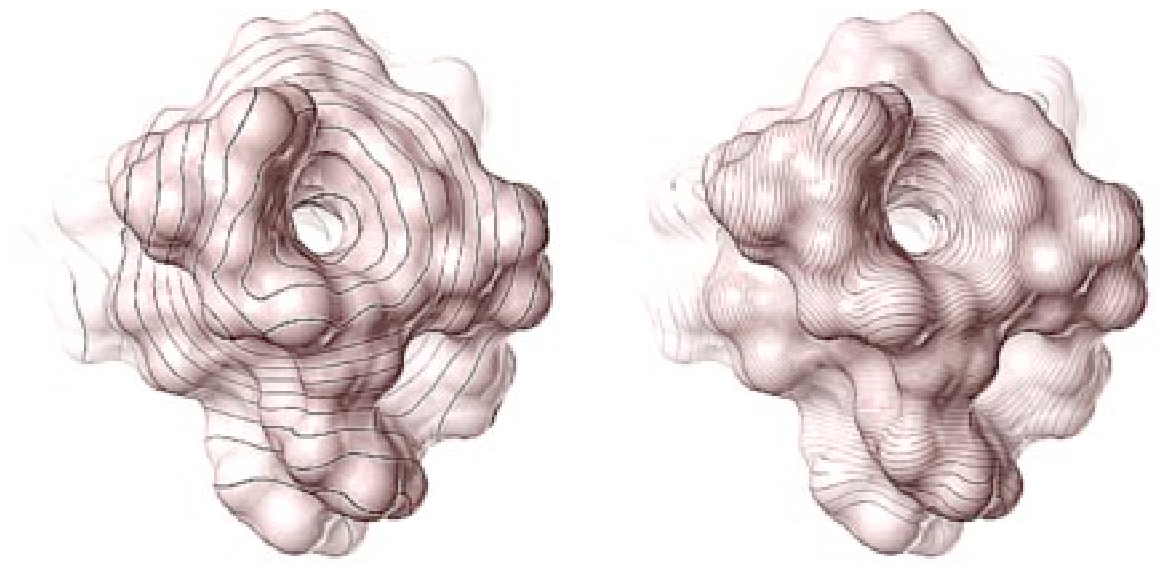


Figure 3: *Isocontour on a molecular surface with different scaling in texture space.*

3.2 Displaying metrics on arbitrary surfaces

An extension of the concept presented in the previous section can be used to display metrics on an arbitrary surface, based on a set of reference planes. Figure 4 demonstrates the application of a 2 D texture to attach tick marks on the solvent accessible surface of a zeolithe.

In contrast to the property based, per vertex binding of texture coordinates, the texture coordinates for the metric texture are generated automatically: the distance of an object vertex to a reference plane is calculated by the harware and on the fly translated to texture coordinates. In this particular case two orthogonal planes are fixed to the orientation of the object's geometry. This type of representation allows for exact measurement of sizes and distance units on a surface.

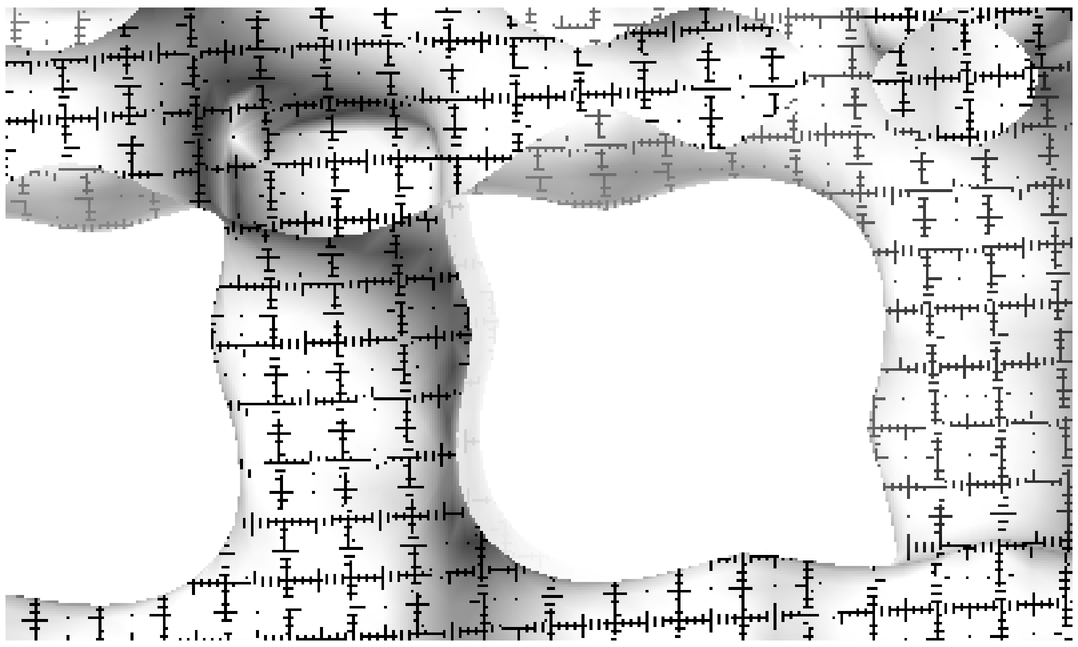


Figure 4: *Display of metrics on a Zeolithe's molecular surface with a 2 D texture.*

3.3 Information filtering

The concept of using a 1 D texture for color coding of surface properties may be extended to 2 D or even 3 D. Thus a maximum of three independent properties can simultaneously be visualized. However, appropriate multidimensional color lookup tables must be designed based on a particular application, because a generalization is either non trivial or eventually impossible. Special care must be taken not to overload the surface with too much information.

One possible, rather general solution can be obtained by combining a 1 D color ramp with a 1 D threshold pattern as presented in the isocontouring example, i.e. color bands are used for one property, whereas orthogonal, discrete isocontour lines code for the second property. In this way it is possible to display two properties simultaneously on the same surface, while still being capable of distinguishing them clearly.

Another approach uses one property to filter the other and display the result on the objects surface, generating additional insight in two different ways: (1) the filter allows the scientist to distinguish between important and irrelevant information, e.g. to display the hot spots on an electrostatic surface potential, or (2) the filter puts an otherwise qualitative property into a quantitative context, e.g., to use the standard deviation from a mean value to provide a hint as to how accurate a represented property actually is at a given location on the object surface.

A good role model for this is the combined display of the electrostatic potential (ESP) and the molecular lipophilic potential (MLP) on the solvent accessible surface of Gramicidine A. The electrostatic potential gives some information on how specific parts of the molecule may interact with other molecules, the molecular lipophilic potential gives a good estimate where the molecule has either contact with water (lipophobic regions) or with the membrane (lipophilic regions). The molecule itself is a channel forming protein, and is loacted in the membrane of bioorganisms, regulating the transport of water molecules and ions. Figure 5 shows the color coding of the solvent accessible surface of Gramicidine A against the ESP filtered with the MLP. The texture used for this example is shown in Figure 8.

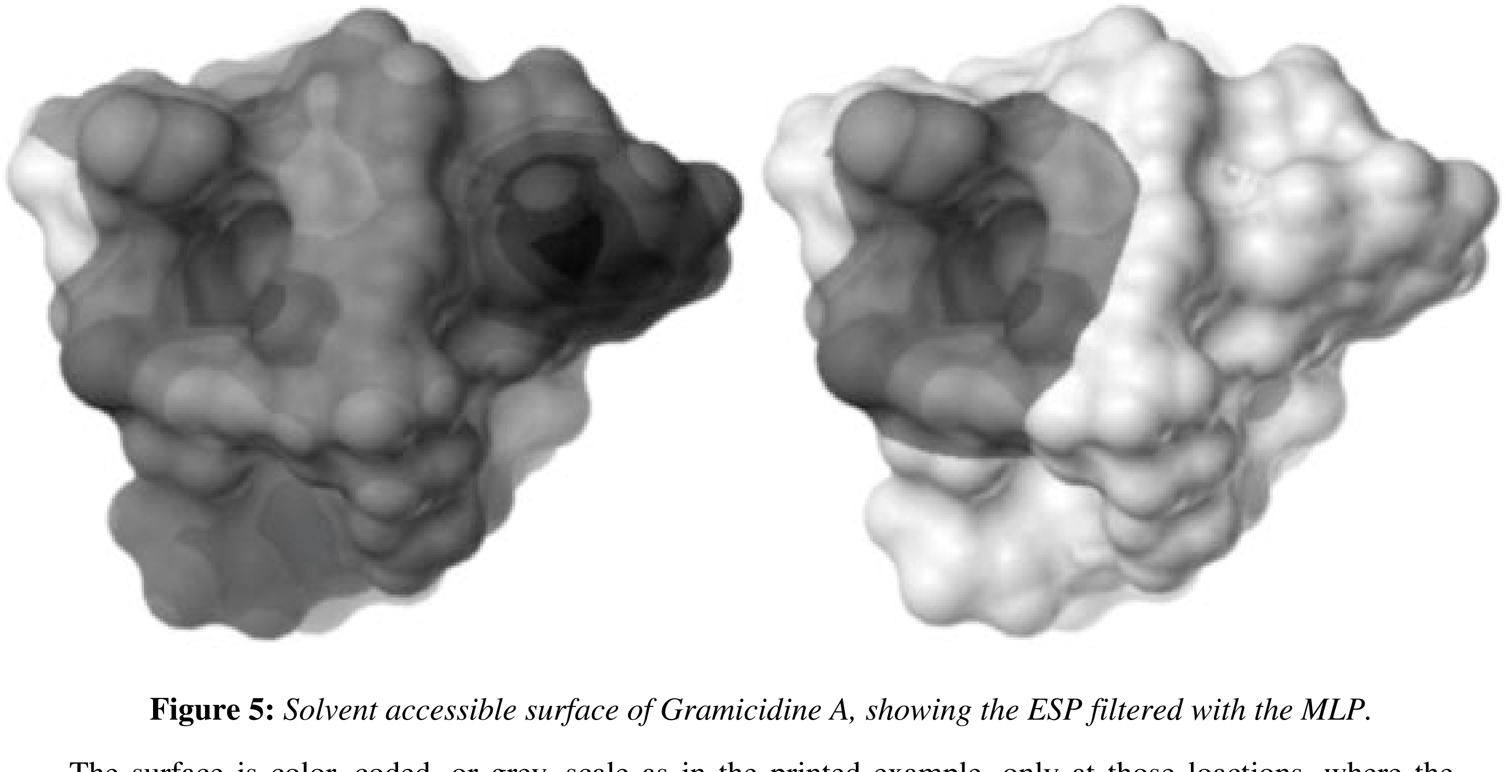


Figure 5: *Solvent accessible surface of Gramicidine A, showing the ESP filtered with the MLP.*

The surface is color coded, or grey scale as in the printed example, only at those loactions, where the surface has a certain lipophobicity. The surface parts with lipophilic behavior are clamped to white. In this example the information is filtered using a delta type function, suppressing all information not exceeding a specified threshold. In other cases, a continouos filter may be more appropriate, to allow a more fine grained quantification.

Another useful application is to filter the electrostatic potential with the electric fileld. Taking the absolute value of the electric field, the filter easily pinpoints the areas of the highest local field gradient, which helps in identifying the binding site of an inhibitor without further interaction of the scientist. With translation in the texture space, one can interactively modify the filter threshold or change the appearance of the color ramp.

3.4 Arbitrary surface clipping

Color coding in the sense of information filtering affects purely the color information of the texture map. By adding transparency as an additional information channel, a lot of flexibility is gained for the comparison of multiple property channels. In a number of cases, transparency even helps in geometrically understanding of a particular property. E.g., the local flexibility of a molecule structure according to the crystallographically determined B factors can be visually represented: the more rigid the structure is, the more opaque the surface will be displayed. Increasing transparency indicates higher floppyness of the domains. Such a transparency map may well be combined with any other color coded property, as it is of interest to study the dynamic properties of a molecule in many different contexts.

An extension to the continuous variation of surface transparency as in the example of molecular flexibility mentioned above is the use of transparency to clip parts of the surface away completely, depending on a property coded into the texture. This can be achieved by setting the alpha values at the appropriate vertices directly to zero. Applied to the information filtering example of Gramicidine A, one can just clip the surface using a texture where all alpha values in the previously white region a set to 0, as is demonstrated in Figure 6.

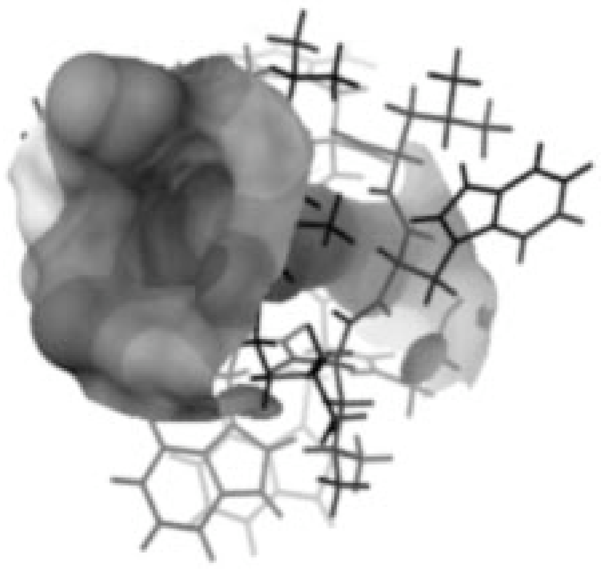


Figure 6: *Clipping of the solvent accessible surface of Gramicidine A according to the MLP.*

There is a distinct advantage in using alpha texture as a component for information filtering: irrelevant information can be completely eliminated, while geometric information otherways hidden within the surface is revealed directly in the context of the surface. And again, it is worthwhile to mention, that by a translation in texture space, the clipping range can be changed interactively!

3.5 Color coding pseudo code example

All above described methods for property visualization on object surfaces are based upon the same texture mapping requirements. Neither are they very demanding in terms of features nor concerning the amount of texture memory needed.

Two options are available to treat texture coordinates that fall outside the range of the parametric unit square. Either the texture can be clamped to constant behaviour, or the entire texture image can be periodically repeated. In the particular examples of 2 D information filtering or property clipping, the parametric s coordinate is used to modify the threshold (clamped), and the t coordinate is used to change the appearance of the color code (repeated). Figure 7 shows different effects of transforming this texture map, while the following pseudo code example expresses the presented texture setup. GL specific calls and constants are highlighted in **boldface**:

```
texParams = {
   TX_MINIFILTER, TX_POINT,
   TX_MAGFILTER,  TX_POINT,
   TX_WRAP_S,     TX_CLAMP,
   TX_WRAP_T,     TX_REPEAT,
   TX_NULL
};

texdef2d(
   texIndex,numTexComponents,
   texWidth,texHeight,texImage,
   numTexParams,texParams
);

texbind(texIndex);
```

The texture image is an array of unsigned integers, where the packing of the data depends on the number of components being used for each texel.

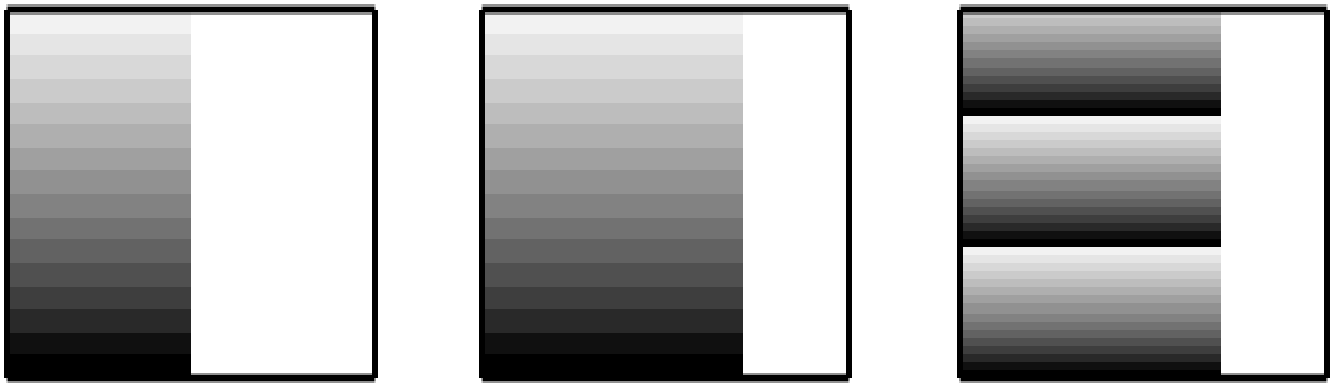


Figure 7: *Example of a 2 D texture used for information filtering, with different transformations applied: original texture (left), translation in s coordinates to adjust filter threshold (middle) and scaling along in t coordinates to change meaning of the texture colors (right).*

The texture environment defines how the texure modifies incoming pixel values. In this case we want to keep the information from the lighting calculation and modulate this with the color coming from the texture image:

```
texEnvParams = {
   TV_MODULATE, TV_NULL
};

tevdef(texEnvIndex,numTexEnvParams,texEnvParams);
tevbind(texEnvIndex);
```

Matrix transformations in texture space must be targeted to a matrix stack that is reserved for texture modifications*:*

```
mmode(MTEXTURE);
   translate(texTransX,0.0,0.0);
   scale(1.0,texScaleY,1.0);
mmode(MVIEWING);
```

The drawing of the object surface requires the binding of a neutral material to get a basic lighting effect. For each vertex, the coordinates, the surface normal and the texture coordinates are traversed in form of calls to **`v3f`**, **`n3f`** and **`t2f`**.

The **`afunction()`** call is only needed in the case of surface clipping. It will prevent the drawing of any part of the polygon that has a texel color with alpha = 0:

```
pushmatrix();
   loadmatrix(modelViewMatrix);
   if(clippingEnabled) {
      afunction(0,AF_NOTEQUAL);
   }
   drawTexturedSurface();
popmatrix();
```

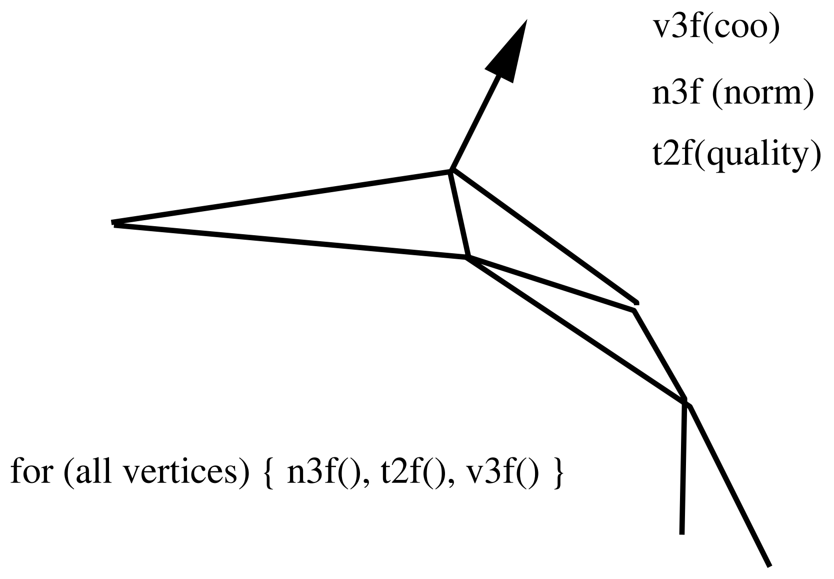


Figure 8: *Schematic representation of the `drawTexturedSurface()` routine.*

4 Real time volume rendering techniques

Volume rendering is a visualization technique used to display 3 D data without an intermediate step of deriving a geometric representation like a solid surface or a chicken wire. The graphical primitives being characteristic for this technique are called voxels, derived from volume element and analog to the pixel. However, voxels describe more than just color, and in fact can represent opacity or shading parameters as well.

A variety of experimental and computational methods produce such volumetric data sets: computer tomography (CT), magnetic resonance imaging (MRI), ultrasonic imaging (UI), confocal light scanning microscopy (CLSM), electron microscopy (EM), X ray crystallography, just to name a few. Characteristic for these data sets are a low signal to noise ratio and a large number of samples, which makes it difficult to use surface based rendering technique, both from a performance and a quality standpoint.

The data structures employed to manipulate volumetric data come in two flavours: (1) the data may be stored as a 3 D grid, or (2) it may be handled as a stack of 2 D images. The former data structure is often used for data that is sampled more or less equally in all the three dimensions, wheras the image stack is preferred with data sets that are high resolution in two dimensions and sparse in the third.

Historically, a wide variety of algorithms has been invented to render volumetric data and range from ray tracing to image compositing [12]. The methods cover an even wider range of performance, where the advantage of image compositing clearly emerges, where several images are created by slicing the volume perpendicular to the viewing axis and then combined back to front, thus summing voxel opacities and colors at each pixel.

In the majority of the cases, the volumetric information is stored using one color channel only. This allows to use lookup tables (LUTs) for alternative color interpretation. I.e., before a particular entry in the color channel is rendered to the frame buffer, the color value is interpreted as a lookup into a table that aliases the original color. By rapidly changing the color and/or opacity transfer function, various structures in the volume are interactively revealed.

By using texture mapping to render the images in the stack, a performance level is reached that is far superior to any other technique used today and allows the real time manipulation of volumetric data. In addition, a considerable degree of flexibility is gained in performing spatial transformations to the volume, since the transformations are applied in the texture domain and cause no performance overhead.

4.1 Volume rendering using 2 D textures

As a linear extension to the original image compositing algotrithm, the 2 D textures can directly replace the images in the stack. A set of mostly quadrilateral polygons is rendered back to front, with each polygon binding its own texture if the depth of the polygon corresponds to the location of the sampled image. Alternatively, polygons inbetween may be textured in a two pass procedure, i.e. the polygon is rendered twice, each time binding one of the two closest images as a texture and filtering it with an appropriate linear weighting factor. In this way, inbetween frames may be obtained even if the graphics subsystem doesn't support texture interpolation in the third dimension.

The resulting volume looks correct as long as the polygons of the image stack are alligned parallel to the screen. However, it is important to be able to look at the volume from arbitrary directions. Because the polygon stack will result in a set of lines when being oriented perpendicular to the screen, a correct perception of the volume is no longer possible.

This problem can easily be soved. By preprocessing the volumetric data into three independent image stacks that are oriented perpendicular to each other, the most appropriate image stack can be selected for rendering based on the orientation of the volume object. I.e., as soon as one stack of textured polygons is rotated towards a critical viewing angle, the rendering function switches to one of the two additional sets of textured polygons, depending on the current orientation of the object.

4.2 Volume rendering using 3 D textures

As described in the previous section, it is not only possible, but almost trivial to implement real time volume rendering using 2 D texture mapping. In addition, the graphics subsystems will operate at peak performance, because they are optimized for fast 2 D texture mapping. However, there are certain limitations to the 2 D texture approach: (1) the memory required by the triple image stack is a factor of three larger than the original data set, which can be critical for large data sets as they are common in medical imaging or microscopy, and (2) the geometry sampling of the volume must be aligned with the 2 D textures concerning the depth, i.e. arbitrary surfaces constructed from a triangle mesh can not easily be colored depending on the properties of a surrounding volume.

For this reason, advanced rendering architectures support hardware implementations of 3 D textures. The correspondence between the volume to be rendered and the 3 D texture is obvious. Any 3 D surface can serve as a sampling device to monitor the coloring of a volumetric property. I.e., the final coloring of the geometry reflects the result of the intersection with the texture. Following this principle, 3 D texture mapping is a fast, accurate and flexible technique for looking at the volume.

The simplest application of 3 D textures is that of a slice plane, which cuts in arbitrary orientations through the volume, which is now represented directly by the texture. The planar polygon being used as geometry in this case will then reflect the contents of the volume as if it were exposed by cutting the object with a knife, as shown in Figure 9: since the transformation of the sampling polygon and that of the 3 D texture is independent, it may be freely oriented within the volume. The property visualized in Figure 9 is the probability of water beeing distributed around a sugar molecule. The orientation of the volume, that means the transformation in the texture space is the same as the molecular structure. Either the molecule, together with the volumetric texture, or the slicing polygon may be reoriented in real time.

An extension of the slice plane approach leads to complete visualization of the entire volume. A stack of slice planes, oriented in parallel to the computer screen, samples the entire 3 D texture. The planes are drawn back to front and in sufficiently small intervals. Geometric transformations of the volume are performed by manipulating the orientation of the texture, keeping the planes in screen parallel orientation, as can be seen in Figure 10, which shows a volume rendered example of a medical application.

This type of volume visualization is greatly enhanced by interactive updates of the color lookup table used to define the texture. In fact a general purpose color ramp editor may be used to vary the lookup colors or the transparency based on the scalar information at a given point in the 3 D volume.

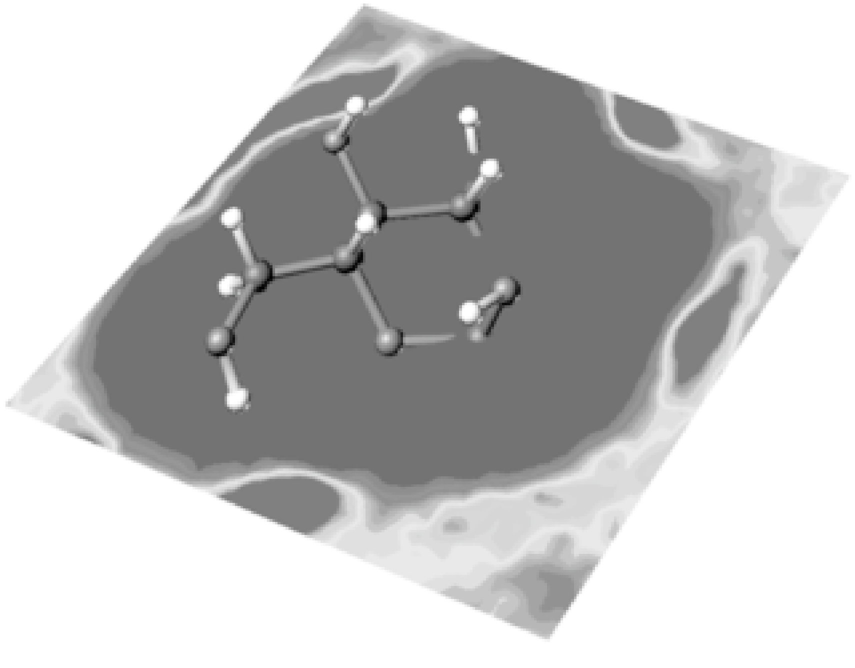


Figure 9: *Slice plane through the water density surrounding a sugar molecule.*

The slice plane concept can be extended to arbitrarily shaped objects. The idea is to probe a volumetric property and to display it wherever the geometric primitives of the probing object cut the volume. The probing geometry can be of any shape, e.g. a sphere, collecting information about the property at a certain distance from a specified point, or it may be extended to describe the surface of an arbitrary object.

The independence of the object's transformation from that of the 3 D texture, offers complete freedom in orienting the surface with respect to the volume. As a further example of a molecular modeling application, this provides an opportunity to look at a molecular surface and have the information about a surrounding volumetric property updated in real time, based on the current orientation of the surface.

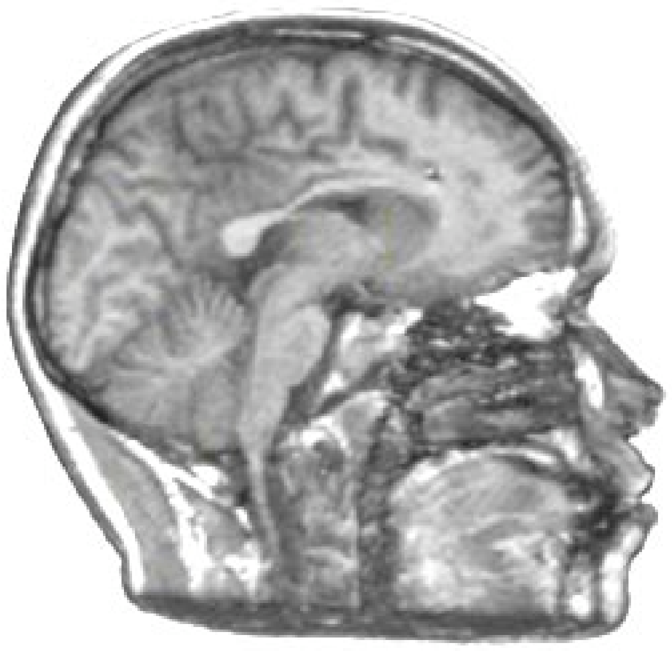


Figure 10: *Volume rendering of MRI data using a stack of screen parallel sectioning planes, which is cut in half to reveal detail in the inner part of the object.*

5 High quality surface rendering

The visualization of solid surfaces with a high degree of local curvature is a major challenge for accurate shading, and where the simple Gouraud shading [13] approach always fails. Here, the lighting calculation is performed for each vertex, depending on the orientation of the surface normal with respect to the light sources. The output of the lighting calculations is an RGB value for the surface vertex. During rasterization of the surface polygon the color value of each pixel is computed by linear interpolation between the vertex colors. Aliasing of the surface highlight is then a consequence of undersampled surface geometry, resulting in moving Gouraud banding patterns on a surface rotating in real time, which is very disturbing. Moreover, the missing accuracy in shading the curved surfaces often leads to a severe loss of information on the object's shape, which is not only critical for the evaluation and analysis of scientific data, but also for the visualization of CAD models, where the visual perception of shape governs the overall design process.

Figure 11 demonstrates this problem using a simple example: on the left, the sphere exhibits typical Gouraud artifacts, on the right the same sphere is shown with a superimposed mesh that reveals the tessellation of the sphere surface. Looking at these images, it is obvious how the shape of the highlight of the sphere was generated from linear interpolation. When rotating the sphere, the highlight begins to oscillate, depending on how near the surface normal at the brightest vertex is with respect to the precise highlight position.

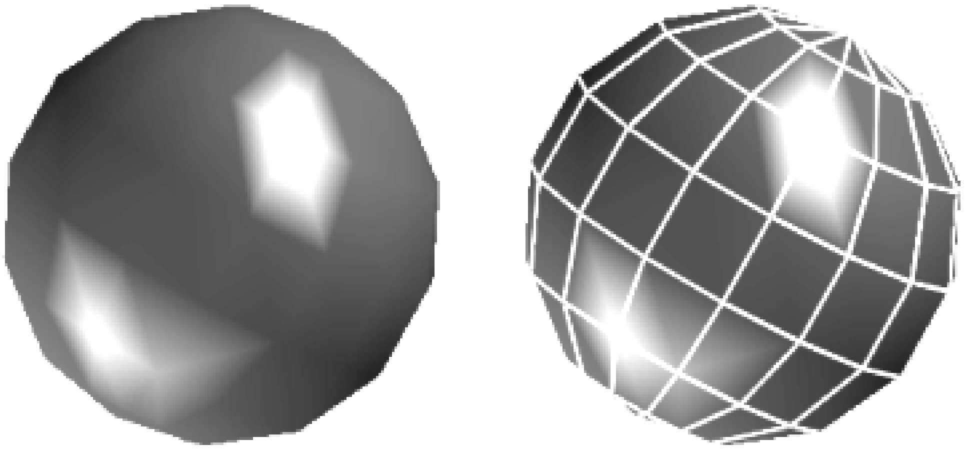


Figure 11: *Gouroud shading artifacts on a moderately tessellated sphere.*

Correct perception of the curvature and constant, non oscillating highlights can only be achieved with computationally much more demanding rendering techniques such as Phong shading [14]. In contrast to linear interpolation of vertex colors, the Phong shading approach interpolates the normal vectors for each pixel of a given geometric primitive, computing the lighting equation in the subsequent step for each pixel. Attempts have been made to overcome some of the computationally intensive steps of the procedure [15], but their performance is insufficient to be a reasonable alternative to Gouraud shading in real time applications.

5.1 Real time Phong shading

With 2 D texture mapping it is now possible to achieve both, high performance drawing speed and highly accurate shading. The resulting picture compares exactly to the surface computed with the complete Phong model with infinite light sources.

The basic idea is to use the image of a high quality rendered sphere as texture. The object's unit length surface normal is interpreted as texture coordinate. Looking at an individual triangle of the polygonal surface, the texture mapping process may be understood as if the image of the perfectly rendered sphere would be wrapped piecewise on the surface polygons. In other words, the surface normal serves as a lookup vector into the texture, acting as a 2 D lookup table that stores precalculated shading information.

The advantage of such a shading procedure is clear: the interpolation is done in texture space and not in RGB, therefore the position of the highlight will never be missed. Note that the tessellation of the texture mapped sphere is exactly the same as for the Gouraud shaded reference sphere in Figure 11.

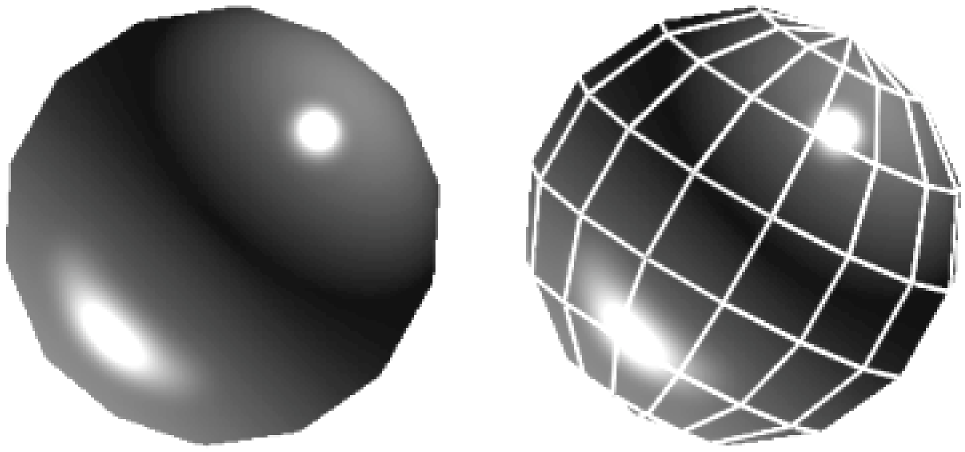


Figure 12: *Phong shaded sphere using surface normals as a lookup for the texture coordinate.*

As previously mentioned, this method of rendering solid surfaces with highest accuracy can be applied to arbitrarily shaped objects. Figure 13 shows the 3 D reconstruction of an electron microscopic experiment, visualizing a large biomolecular complex, the asymmetric unit membrane of the urinary bladder. The difference between Gouraud shading and the texture mapping implementation of Phong shading is obvious, and for the sake of printing quality, can be seen best when looking at the closeups. Although this trick is so far only applicable for infinitely distant light sources, it is a tremendous aid for the visualization of highly complex surfaces.

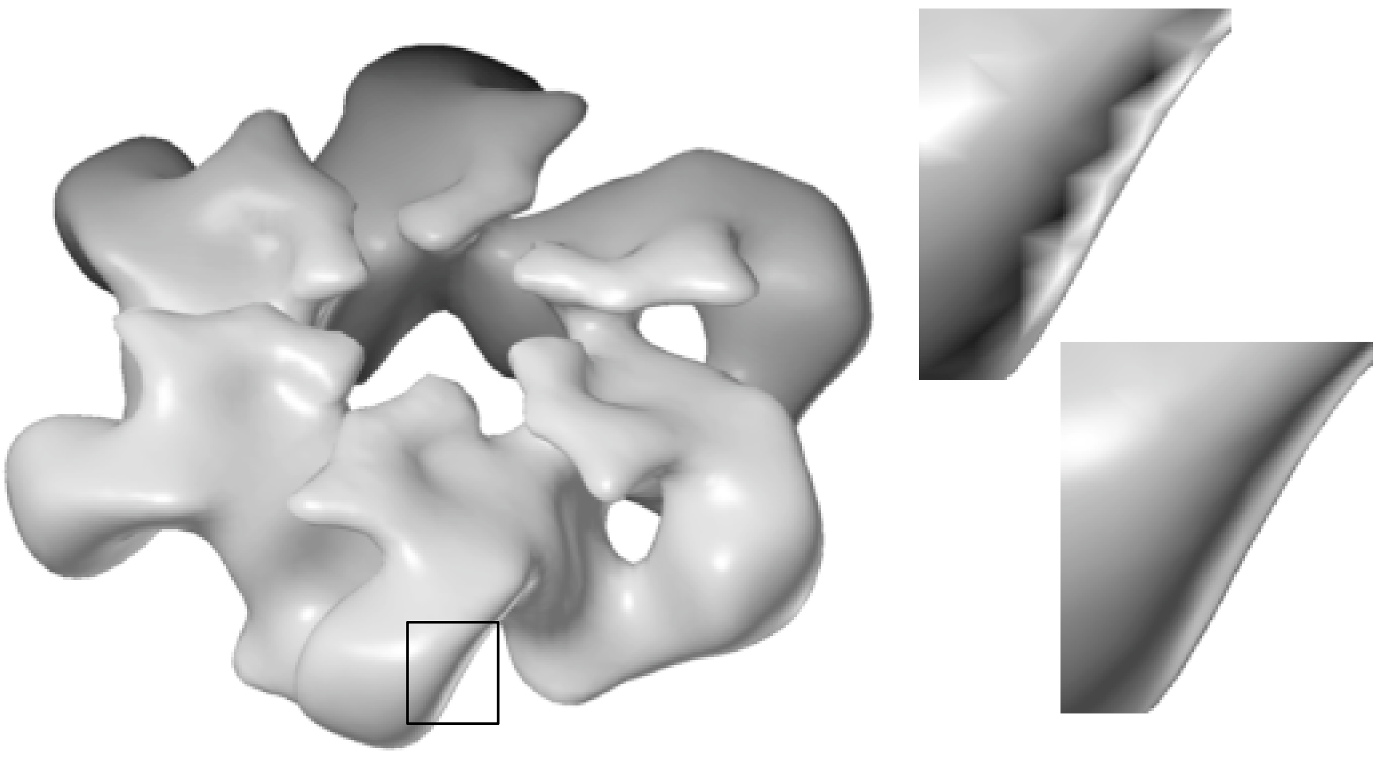


Figure 13: *Application of the texture mapped Phong shading to a complex surface representing a biomolecular structure. The closeups demonstrate the difference between Gouraud shading (above right) and Phong shading (below right) when implemented using texture mapping*

5.2 Phong shading pseudo code example

The setup for the texture mapping as used for Phong shading is shown in the following code fragment:

```
texParams = {
   TX_MINIFILTER, TX_POINT,
   TX_MAGFILTER,  TX_BILINEAR,
   TX_NULL
};

texdef2d(
   texIndex,numTexComponents,
   texWidth,texHeight,texImage,
   numTexParams,texParams
);
```

```
texbind(texIndex);

texEnvParams = { TV_MODULATE, TV_NULL };

tevdef(texEnvIndex,numTexEnvParams,texEnvParams);
tevbind(texEnvIndex);
```

As texture, we can use any image of a high quality rendered sphere either with RGB or one intensity component only. The RGB version allows the simulation of light sources with different colors.

The most important change for the vertex calls in this model is that we do not pass the surface normal data with the **n3f** command as we normally do when using Gouraud shading. The normal is passed as texture coordinate and therefore processed with the **t3f** command.

Surface normals are transformed with the current model view matrix, although only rotational components are considered. For this reason the texture must be aligned with the current orientation of the object. Also, the texture space must be scaled and shifted to cover a circle centered at the origin of the s/t coordinate system, with a unit length radius to map the surface normals:

```
mmode(MTEXTURE);
   loadmatrix(identityMatrix);
   translate(0.5,0.5,0.0);
   scale(0.5,0.5,1.0);
   multmatrix(rotationMatrix);
mmode(MVIEWING);

drawTexPhongSurface();
```

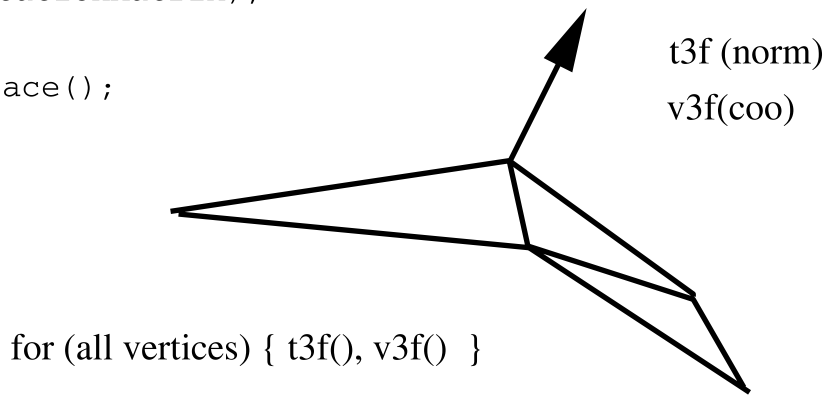


Figure 15: *Schematic representation of the `drawTexPhongSurface()` routine.*

6 Conclusions

Silicon Graphics has recently introduced a new generation of graphics subsystems, which support a variety of texture mapping techniques in hardware without performance penalty. The potential of using this technique in technical, scientific and engineering visualization applications has been demonstrated.

Hardware supported texture mapping offers solutions to important visualization problems that have either not been solved yet or did not perform well enough to enter the world of interactive graphics applications. Although most of the examples presented here could be implemented using techniques other than texture mapping, the tradeoff would either be complete loss of performance or an unmaintainable level of algorithmic complexity.

Most of the examples were taken from the molecular modelling market, where one has learned over the years to handle complex 3 D scenarios interactively and in an analytic manner. What has been shown here can also be applied in other areas of scientific, technical or engineering visualization. With the examples shown in this report, it should be possible for software engineers developing application software in other markets to use the power and flexibility of texture mapping and to adapt the shown solutions to their specific case.

One important, general conclusion may be drawn from this work: one has to leave the traditional mind set about texture mapping and go back to the basics in order to identify the participating components and to understand their generic role in the procedure. Once this step is done it is very simple to use this technique in a variety of visualization problems.

All examples were implemented on a Silicon Graphics Crimson Reality Engine [7] equipped with two raster managers. The programs were written in C, either in mixed mode GLX or pure GL.

7 References

[1] Blinn, J.F. and Newell, M.E. *Texture and reflection in computer generated images*, Communications of the ACM 1976, **19**, 542 547.

[2] Blinn, J.F. *Simulation of wrinkled surfaces* Computer Graphics 1978, **12**, 286 292.

[3] Haeberli, P. and Segal, M. *Texture mapping as a fundamental drawing primitive*, Proceedings of the fourth eurographics workshop on rendering, 1993, 259 266.

[4] Peachy, D.R. *Solid texturing of complex surfaces*, Computer Graphics 1985, **19**, 279 286.

[5] Gardner, G.Y. *Simulation of natural scenes using textured quadric surfaces,* Computer Graphics 1984, **18**, 11 20.

[6] Gardner, G.Y. *Visual simulations of clouds*, Computer Graphics 1985, **19,** 279 303.

[7] Akeley, K. *Reality Engine Graphics*, Computer Graphics 1993, **27**, 109 116.

[8] Catmull, E.A. *Subdivision algorithm for computer display of curved surfaces,* Ph.D. thesis University of Utah, 1974.

[9] Crow, F.C. *Summed area tables for texture mapping*, Computer Graphics 1984, **18**, 207 212.

[10] Dill, J.C. *An application of color graphics to the display of surface curvature*, Computer Graphics 1981, **15**, 153 161.

[11] Sabella, P. *A rendering algorithm for visualizing 3d scalar fields,* Computer Graphics, 1988 **22**, 51 58.

[12] Drebin, R. Carpenter, L. and Hanrahan, P. *Volume Rendering,* Computer Graphics, 1988, **22**, 65 74.

[13] Gouraud, H. *Continuous shading of curved surfaces*, IEEE Transactions on Computers, 1971, **20**, 623 628.

[14] Phong, B.T. *Illumination for computer generated pictures,* Communications of the ACM 1978, **18**, 311 317.

[15] Bishop, G. and Weimer, D.M. *Fast Phong shading*, Computer Graphics, 1986, **20**, 103 106.

Efficient Bump Mapping Hardware

Mark Peercy
John Airey
Brian Cabral
Silicon Graphics Computer Systems [*]

Abstract

We present a bump mapping method that requires minimal hardware beyond that necessary for Phong shading. We eliminate the costly per-pixel steps of reconstructing a tangent space and perturbing the interpolated normal vector by a) interpolating vectors that have been transformed into tangent space at polygon vertices and b) storing a precomputed, perturbed normal map as a texture. The savings represents up to a factor of two in hardware or time compared to a straightforward implementation of bump mapping.

CR categories and subject descriptors: I.3.3 [Computer Graphics]: Picture/Image generation; I.3.7 [Image Processing]: Enhancement

Keywords: hardware, shading, bump mapping, texture mapping.

1 INTRODUCTION

Shading calculations in commercially available graphics systems have been limited to lighting at the vertices of a set of polygons, with the resultant colors interpolated and composited with a texture. The drawbacks of Gouraud interpolation [9] are well known and include diffused, crawling highlights and mach banding. The use of this method is motivated primarily by the relatively large cost of the lighting computation. When done at the vertices, this cost is amortized over the interiors of polygons.

The division of a computation into per-vertex and per-pixel components is a general strategy in hardware graphics acceleration [1]. Commonly, the vertex computations are performed in a general floating point processor or cpu, while the per-pixel computations are in special purpose, fixed point hardware. The division is a function of cost versus the general applicability, in terms of quality and speed, of a feature. Naturally, the advance of processor and application-specific integrated circuit technology has an impact on the choice.

Because the per-vertex computations are done in a general processor, the cost of a new feature tends to be dominated by additional per-pixel hardware. If this feature has a very specific application, the extra hardware is hard to justify because it lays idle in applications that do not leverage it. And in low-end or game systems, where every transistor counts, additional rasterization hardware is particularly expensive. An alternative to extra hardware is the reuse of existing hardware, but this option necessarily runs much slower.

[*] {peercy,airey,cabral}@sgi.com
2011 N. Shoreline Boulevard
Mountain View, California 94043-1389

Shading quality can be increased dramatically with Phong shading [13], which interpolates and normalizes vertex normal vectors at each pixel. Light and halfangle vectors are computed directly in world space or interpolated, either of which requires their normalization for a local viewer and light. Figure 1 shows rasterization hardware for one implementation of Phong shading, upon which we base this discussion.[1] This adds significant cost to rasterization hardware. However higher quality lighting is almost universally desired in three-dimensional graphics applications, and advancing semiconductor technology is making Phong shading hardware more practical. We take Phong shading and texture mapping hardware as a prerequisite for bump mapping, assuming they will be standard in graphics hardware in the future.

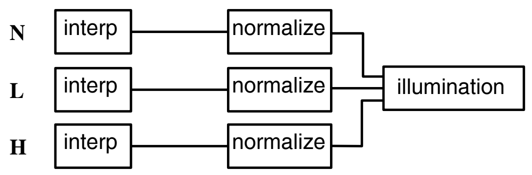


Figure 1. One implementation of Phong shading hardware.

Bump mapping [3] is a technique used in advanced shading applications for simulating the effect of light reflecting from small perturbations across a surface. A single component texture map, $f(u,v)$, is interpreted as a height field that perturbs the surface along its normal vector, $\mathbf{N} = (\mathbf{P}_u \times \mathbf{P}_v)/|(\mathbf{P}_u \times \mathbf{P}_v)|$, at each point. Rather than actually changing the surface geometry, however, only the normal vector is modified. From the partial derivatives of the surface position in the u and v parametric directions ($\mathbf{P}_u$ and $\mathbf{P}_v$), and the partial derivatives of the image height field in u and v (f_u and f_v), a perturbed normal vector $\mathbf{N}'$ is given by [3]:

$$\mathbf{N}' = ((\mathbf{P}_u \times \mathbf{P}_v) + \mathbf{D})/|(\mathbf{P}_u \times \mathbf{P}_v) + \mathbf{D}| \qquad (1)$$

where

$$\mathbf{D} = -f_u(\mathbf{P}_v \times \mathbf{N}) - f_v(\mathbf{N} \times \mathbf{P}_u) \qquad (2)$$

In these equations, $\mathbf{P}_u$ and $\mathbf{P}_v$ are not normalized. As Blinn points out [3], this causes the bump heights to be a function of the surface scale because $\mathbf{P}_u \times \mathbf{P}_v$ changes at a different rate than $\mathbf{D}$. If the surface scale is doubled, the bump heights are halved. This dependence on the surface often is an undesirable feature, and Blinn suggests one way to enforce a constant bump height.

A full implementation of these equations in a rasterizer is impractical, so the computation is divided among a preprocessing step, per-vertex, and per-pixel calculations. A natural method to implement bump mapping in hardware, and one that is planned for a high-end graphics workstation [6], is to compute $\mathbf{P}_u \times \mathbf{P}_v$, $\mathbf{P}_v \times \mathbf{N}$, and $\mathbf{N} \times \mathbf{P}_u$ at polygon vertices and interpolate them to polygon interiors. The perturbed normal vector is computed and normalized as in Equation 1, with f_u and f_v read from a texture map. The resulting normal vector is used in an illumination model.

The hardware for this method is shown in Figure 2. Because $\mathbf{P}_u$

[1] Several different implementations of Phong shading have been suggested [11][10][4][5][7][2] with their own costs and benefits. Our bump mapping algorithm can leverage many variations, and we use this form as well as Blinn's introduction of the halfangle vector for clarity.

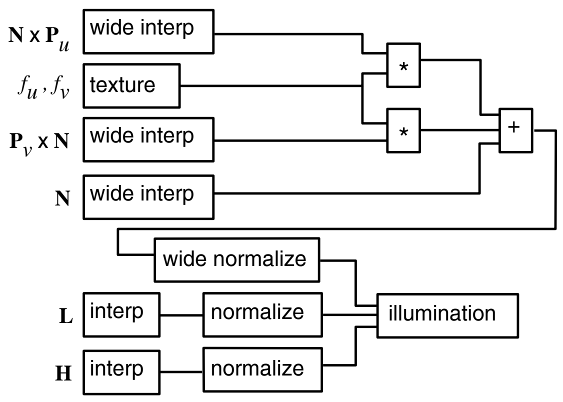


Figure 2. A suggested implementation of bump mapping hardware.

and $\mathbf{P}_v$ are unbounded, the three interpolators, the vector addition, vector scaling, and normalization must have much greater range and precision than those needed for bounded vectors. These requirements are noted in the figure. One approximation to this implementation has been been proposed [8], where $\mathbf{P}_v \times \mathbf{N}$ and $\mathbf{N} \times \mathbf{P}_u$ are held constant across a polygon. While avoiding their interpolation, this approximation is known to have artifacts [8].

We present an implementation of bump mapping that leverages Phong shading hardware at full speed, eliminating either a large investment in special purpose hardware or a slowdown during bump mapping. The principal idea is to transform the bump mapping computation into a different reference frame. Because illumination models are a function of vector operations (such as the dot product) between the perturbed normal vector and other vectors (such as the light and halfangle), they can be computed relative to any frame. We are able to push portions of the bump mapping computation into a preprocess or the per-vertex processor and out of the rasterizer. As a result, minimal hardware is added to a Phong shading circuit.

2 OUR BUMP-MAPPING ALGORITHM

We proceed by recognizing that the original bump mapping approximation [3] assumes a surface is locally flat at each point. The perturbation is, therefore, a function only of the local tangent space. We define this space by the normal vector, $\mathbf{N}$, a tangent vector, $\mathbf{T} = \mathbf{P}_u/|\mathbf{P}_u|$, and a binormal vector, $\mathbf{B} = (\mathbf{N} \times \mathbf{T})$. $\mathbf{T}$, $\mathbf{B}$, and $\mathbf{N}$ form an orthonormal coordinate system in which we perform the bump mapping. In this space, the perturbed normal vector is (see appendix):

$$\mathbf{N}'_{TS} = (a, b, c)/\sqrt{a^2 + b^2 + c^2} \tag{3}$$

$$a = -f_u(\mathbf{B} \cdot \mathbf{P}_v) \tag{4}$$

$$b = -(f_v|\mathbf{P}_u| - f_u(\mathbf{T} \cdot \mathbf{P}_v)) \tag{5}$$

$$c = |\mathbf{P}_u \times \mathbf{P}_v| \tag{6}$$

The coefficients a, b, and c are a function of the surface itself (via $\mathbf{P}_u$ and $\mathbf{P}_v$) and the height field (via f_u and f_v). Provided that the bump map is fixed to a surface, the coefficients can be precomputed for that surface at each point of the height field and stored as a texture map (we discuss approximations that relax the surface dependence below). The texel components lie in the range -1 to 1.

The texture map containing the perturbed normal vector is filtered as a simple texture using, for instance, tri-linear mipmap filtering. The texels in the coarser levels of detail can be computed by filtering finer levels of detail and renormalizing or by filtering the height field and computing the texels directly from Equations 3-6. It is well known that this filtering step tends to average out the bumps at large minifications, leading to artifacts at silhouette edges. Proper filtering of bump maps requires computing the reflected radiance over all bumps contributing to a single pixel, an option that is not practical for hardware systems. It should also be noted that, after mipmap interpolation, the texture will not be normalized, so we must normalize it prior to lighting.

For the illumination calculation to proceed properly, we transform the light and halfangle vectors into tangent space via a 3×3 matrix whose columns are $\mathbf{T}$, $\mathbf{B}$, and $\mathbf{N}$. For instance, the light vector, $\mathbf{L}$, is transformed by

$$\mathbf{L}_{TS} = \mathbf{L} \begin{pmatrix} \mathbf{T} & \mathbf{B} & \mathbf{N} \\ \downarrow & \downarrow & \downarrow \end{pmatrix} \tag{7}$$

Now the diffuse term in the illumination model can be computed from the perturbed normal vector from the texture map and the transformed light: $\mathbf{N}'_{TS} \cdot \mathbf{L}_{TS}$. The same consideration holds for the other terms in the illumination model.

The transformations of the light and halfangle vectors should be performed at every pixel; however, if the change of the local tangent space across a polygon is small, a good approximation can be obtained by transforming the vectors only at the polygon vertices. They are then interpolated and normalized in the polygon interiors. This is frequently a good assumption because tangent space changes rapidly in areas of high surface curvature, and an application will need to tessellate the surfaces more finely in those regions to reduce geometric faceting.

This transformation is, in spirit, the same as one proposed by Kuijk and Blake to reduce the hardware required for Phong shading [11]. Rather than specifying a tangent and binormal explicitly, they rotate the reference frames at polygon vertices to orient all normal vectors in the same direction (such as $(0, 0, 1)$). In this space, they no longer interpolate the normal vector (an approximation akin to ours that tangent space changes slowly). If the bump map is identically zero, we too can avoid an interpolation and normalization, and we will have a result similar to their approximation. It should be noted that the highlight in this case is slightly different than that obtained by the Phong circuit of Figure 1, yet it is still phenomenologically reasonable.

The rasterization hardware required for our bump mapping algorithm is shown in Figure 3; by adding a multiplexer to the Phong shading hardware of Figure 1, both the original Phong shading and bump mapping can be supported. Absent in the implementation of Figure 2, this algorithm requires transforming the light and halfangle vectors into tangent space at each vertex, storing a three-component texture map instead of a two-component map, and having a separate map for each surface. However, it requires only a multiplexer beyond Phong shading, avoids the interpolation of $(\mathbf{P}_v \times \mathbf{N})$ and $(\mathbf{N} \times \mathbf{P}_u)$, the perturbation of the normal vector at each pixel, and the extended range and precision needed for arithmetic on unbounded vectors. Effectively, we have traded per-pixel calculations cast in hardware for per-vertex calculations done in the general geometry processor. If the application is limited by the rasterization, it will run at the same speed with bump mapping as with Phong shading.

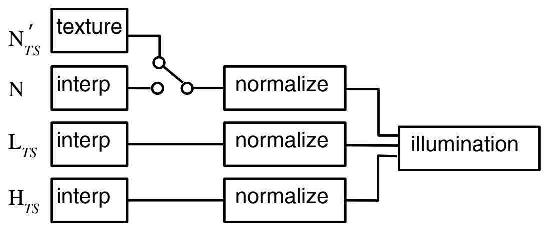


Figure 3. One implementation of our bump mapping algorithm.

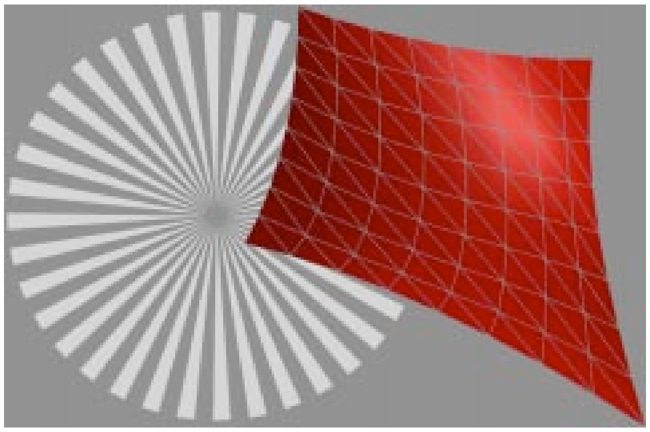


Figure 4.The pinwheel height field is used as a bump map for the tesselated, bicubic surface.

2.1 Object-Space Normal Map

If the texture map is a function of the surface parameterization, another implementation is possible: the lighting model can be computed in object space rather than tangent space. Then, the texture stores the perturbed normal vectors in object space, and the light and halfangle vectors are transformed into object space at the polygon vertices and interpolated. Thus, the matrix transformation applied to the light and halfangle vectors is shared by all vertices, rather than one transformation for each vertex. This implementation keeps the rasterization hardware of Figure 3, significantly reduces the overhead in the geometry processor, and can coexist with the first formulation.

2.2 Removing the surface dependence

The primary drawback of our method is the surface dependence of the texture map. The dependence of the bumps on surface scale is shared with the traditional formulation of bump mapping. Yet in addition, our texture map is a function of the surface, so the height field can not be shared among surfaces with different parameterizations. This is particularly problematic when texture memory is restricted, as in a game system, or during design when a bump map is placed on a new surface interactively.

All of the surface dependencies can be eliminated under the assumption that, locally, the parameterization is the same as a square patch (similar to, yet more restrictive than, the assumption Blinn makes in removing the scale dependence [3]). Then, $\mathbf{P}_u$ and $\mathbf{P}_v$ are orthogonal ($\mathbf{P}_u \cdot \mathbf{P}_v = \mathbf{T} \cdot \mathbf{P}_v = 0$) and equal in magnitude ($|\mathbf{P}_u| = |\mathbf{P}_v|$). To remove the bump dependence on surface scale,

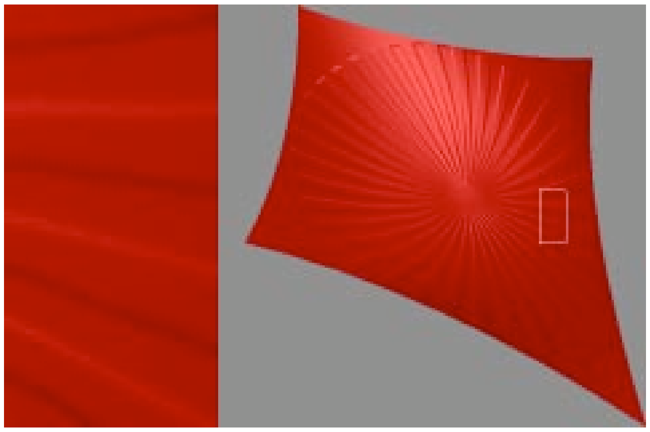


Figure 6.Bump mapping with the hardware in Figure 3, and the texture map from Eqns 3-6.

we simply choose $|\mathbf{P}_u| = |\mathbf{P}_v| = k$, where k is a constant giving a relative height of the bumps. This, along with the orthogonality condition, reduce Equations 3-6 to

$$\mathbf{N}'_{TS} = (a, b, c)/\sqrt{a^2 + b^2 + c^2} \qquad (8)$$

$$\begin{aligned} a &= -kf_u && (9) \\ b &= -kf_v && (10) \\ c &= k^2 && (11) \end{aligned}$$

The texture map becomes a function only of the height field and not of the surface geometry, so it can be precomputed and used on any surface.

The square patch assumption holds for several important surfaces, such as spheres, tori, surfaces of revolution, and flat rectangles. In addition, the property is highly desirable for general surfaces because the further $\mathbf{P}_u$ and $\mathbf{P}_v$ are from orthogonal and equal in magnitude, the greater the warp in the texture map when applied to a surface. This warping is typically undesirable, and its elimination has been the subject of research [12]. If the surface is already reasonably parameterized or can be reparameterized, the approximation in Equations 8-11 is good.

3 EXAMPLES

Figures 5-7 compare software simulations of the various bump mapping implementations. All of the images, including the height field, have a resolution of 512x512 pixels. The height field, Figure 4, was

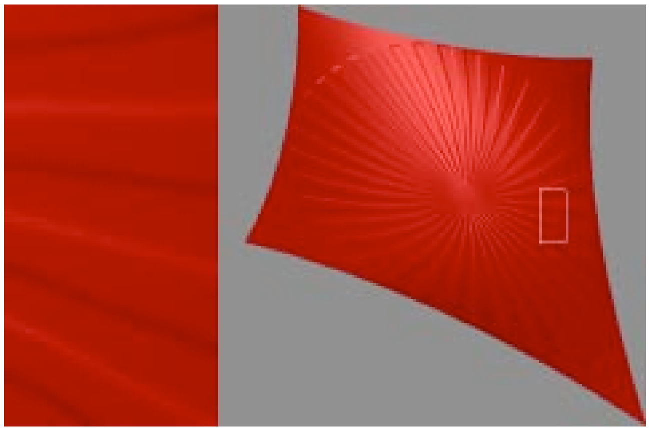


Figure 4. Bump mapping using the hardware implementation shown in Figure 2.

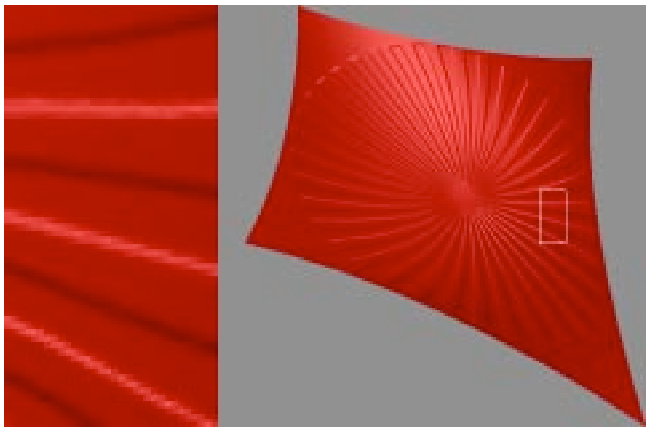


Figure 7.Bump mapping with the hardware in Figure 3, and the texture map from Eqns 8-11.

chosen as a pinwheel to highlight filtering and implementation artifacts, and the surface, Figure 4, was chosen as a highly stretched bicubic patch subdivided into 8x8x2 triangles to ensure that $\mathbf{P}_u$ and $\mathbf{P}_v$ deviate appreciably from orthogonal. The texture maps were filtered with trilinear mipmapping.

Figure 5 shows the image computed from the implementation of bump mapping from Figure 2. The partial derivatives, f_u and f_v, in this texture map and the others were computed with the derivative of a Gaussian covering seven by seven samples.

Figures 6 and 7 show our implementation based on the hardware of Figure 3; they differ only in the texture map that is employed. Figure 6 uses a texture map based on Equations 3-6. Each texel was computed from the analytic values of $\mathbf{P}_u$ and $\mathbf{P}_v$ for the bicubic patch. The difference between this image and Figure 5 is almost imperceptible, even under animation, as can be seen in the enlarged insets. The texture map used in Figure 7 is based on Equations 8-11, where the surface dependence has been removed. Minor differences can be seen in the rendered image compared to Figures 5 and 6; some are visible in the inset. All three implementations have similar filtering qualities and appearance during animation.

4 DISCUSSION

We have presented an implementation of bump mapping that, by transforming the lighting problem into tangent space, avoids any significant new rasterization hardware beyond Phong shading. To summarize our algorithm, we

- precompute a texture of the perturbed normal in tangent space
- transform all shading vectors into tangent space per vertex
- interpolate and renormalize the shading vectors
- fetch and normalize the perturbed normal from the texture
- compute the illumination model with these vectors

Efficiency is gained by moving a portion of the problem to the vertices and away from special purpose bump mapping hardware in the rasterizer; the incremental cost of the per-vertex transformations is amortized over the polygons.

It is important to note that the method of transforming into tangent space for bump mapping is independent of the illumination model, provided the model is a function only of vector operations on the normal. For instance, the original Phong lighting model, with the reflection vector and the view vector for the highlight, can be used instead of the halfangle vector. In this case, the view vector is transformed into tangent space and interpolated rather than the halfangle. As long as all necessary shading vectors for the illumination model are transformed into tangent space and interpolated, lighting is proper.

Our approach is relatively independent of the particular implementation of Phong shading, however it does require the per-pixel illumination model to accept vectors rather than partial illumination results. We have presented a Phong shading circuit where almost no new hardware is required, but other implementations may need extra hardware. For example, if the light and halfangle vectors are computed directly in eye space, interpolators must be added to support our algorithm. The additional cost still will be very small compared to a straightforward implementation.

Phong shading likely will become a standard addition to hardware graphics system because of its general applicability. Our algorithm extends Phong shading in such an effective manner that it is natural to support bump mapping even on the lowest cost Phong shading systems.

5 ACKNOWLEDGEMENTS

This work would not have been possible without help, ideas, conversations and encouragement from Pat Hanrahan, Bob Drebin, Kurt Akeley, Erik Lindholm and Vimal Parikh. Also thanks to the anonymous reviewers who provided good and insightful suggestions.

APPENDIX

Here we derive the perturbed normal vector in tangent space, a reference frame given by tangent, $\mathbf{T} = \mathbf{P}_u/|\mathbf{P}_u|$; binormal, $\mathbf{B} = (\mathbf{N} \times \mathbf{T})$; and normal, $\mathbf{N}$, vectors. $\mathbf{P}_v$ is in the plane of the tangent and binormal, and it can be written:

$$\mathbf{P}_v = (\mathbf{T} \cdot \mathbf{P}_v)\mathbf{T} + (\mathbf{B} \cdot \mathbf{P}_v)\mathbf{B} \tag{12}$$

Therefore

$$\mathbf{P}_v \times \mathbf{N} = (\mathbf{B} \cdot \mathbf{P}_v)\mathbf{T} - (\mathbf{T} \cdot \mathbf{P}_v)\mathbf{B} \tag{13}$$

The normal perturbation (Equation 2) is:

$$\begin{aligned} \mathbf{D} &= -f_u(\mathbf{P}_v \times \mathbf{N}) - f_v|\mathbf{P}_u|\mathbf{B} && (14) \\ &= -f_u(\mathbf{B} \cdot \mathbf{P}_v)\mathbf{T} - (f_v|\mathbf{P}_u| - f_u(\mathbf{T} \cdot \mathbf{P}_v))\mathbf{B} && (15) \end{aligned}$$

Substituting the expression for $\mathbf{D}$ and $\mathbf{P}_u \times \mathbf{P}_v = |\mathbf{P}_u \times \mathbf{P}_v|\mathbf{N}$ into Equation 1, normalizing, and taking $\mathbf{T}_{TS} = (1,0,0)$, $\mathbf{B}_{TS} = (0,1,0)$, and $\mathbf{N}_{TS} = (0,0,1)$ leads directly to Equations 3-6.

References

[1] Akeley, K. RealityEngine graphics. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (Aug. 1993), J. T. Kajiya, Ed., vol. 27, pp. 109–116.

[2] Bishop, G., and Weimer, D. M. Fast Phong shading. In *Computer Graphics (SIGGRAPH '86 Proceedings)* (Aug. 1986), D. C. Evans and R. J. Athay, Eds., vol. 20, pp. 103–106.

[3] Blinn, J. F. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)* (Aug. 1978), vol. 12, pp. 286–292.

[4] Claussen, U. Real time phong shading. In *Fifth Eurographics Workshop on Graphics Hardware* (1989), D. Grimsdale and A. Kaufman, Eds.

[5] Claussen, U. On reducing the phong shading method. *Computers and Graphics 14*, 1 (1990), 73–81.

[6] Cosman, M. A., and Grange, R. L. CIG scene realism: The world tomorrow. In *Proceedings of I/ITSEC 1996 on CD-ROM* (1996), p. 628.

[7] Deering, M. F., Winner, S., Schediwy, B., Duffy, C., and Hunt, N. The triangle processor and normal vector shader: A VLSI system for high performance graphics. In *Computer Graphics (SIGGRAPH '88 Proceedings)* (Aug. 1988), J. Dill, Ed., vol. 22, pp. 21–30.

[8] Ernst, I., Jackel, D., Russeler, H., and Wittig, O. Hardware supported bump mapping: A step towards higher quality real-time rendering. In *10th Eurographics Workshop on Graphics Hardware* (1995), pp. 63–70.

[9] Gouraud, H. Computer display of curved surfaces. *IEEE Trans. Computers C-20*, 6 (1971), 623–629.

[10] Jackel, D., and Russeler, H. A real time rendering system with normal vector shading. In *9th Eurographics Workshop on Graphics Hardware* (1994), pp. 48–57.

[11] Kuijk, A. A. M., and Blake, E. H. Faster phong shading via angular interpolation. *Computer Graphics Forum 8*, 4 (Dec. 1989), 315–324.

[12] Maillot, J., Yahia, H., and Verroust, A. Interactive texture mapping. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (Aug. 1993), J. T. Kajiya, Ed., vol. 27, pp. 27–34.

[13] Phong, B.-T. Illumination for computer generated pictures. *Communications of the ACM 18*, 6 (June 1975), 311–317.

Fitting Virtual Lights For Non-Diffuse Walkthroughs

Bruce Walter Gün Alpay Eric Lafortune Sebastian Fernandez Donald P. Greenberg

Cornell Program of Computer Graphics *

Abstract

This paper describes a technique for using a simple shading method, such as the Phong lighting model, to approximate the appearance calculated by a more accurate method. The results are then suitable for rapid display using existing graphics hardware and portable via standard graphics API's. Interactive walkthroughs of view-independent non-diffuse global illumination solutions are explored as the motivating application.

CR Categories: I.3.7 [Computer Graphics]: Three Dimensional Graphics and Realism—Shading

Keywords: interactive walkthroughs, non-diffuse appearance, global illumination, Phong shading

1 INTRODUCTION

This paper describes a method to take a view-independent non-diffuse global illumination solution and approximate it in a form that is suitable for rapid display and interactive walkthroughs. The method fits "virtual lights" to each object that, when displayed using a simple Phong lighting model, will closely reproduce its correct appearance.

One goal of realistic computer graphics is to let a viewer experience a virtual space as if they were physically present in a real space. There are many possible aspects to this mimicry, but here we will emphasize two facets. We want the viewer to be able to move about and explore the space in a natural and unrestricted way, and we want to match the appearance of the real space as closely as possible.

Real lighting is complex and subtle. Global illumination calculations are necessary if we hope to duplicate its appearance. These calculations are expensive, but if we are willing to restrict ourselves to a static environment, this part of the simulation can done as a pre-process. However, we still need to display the results rapidly if we want interactive walkthroughs. To accomplish this, we would like to leverage the existing 3D graphics hardware/software infrastructure.

Unfortunately, there is no standard format for storing non-diffuse lighting information; previously this has meant displaying a diffuse-only approximation to the actual appearance. While the results can be impressive, the absence of directionally dependent lighting effects, such as glossy highlights, means that important perceptual cues are missing.

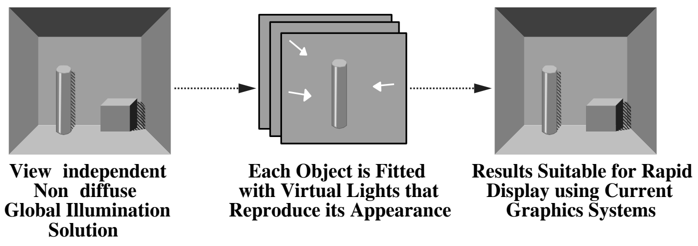

View independent Non diffuse Global Illumination Solution
Each Object is Fitted with Virtual Lights that Reproduce its Appearance
Results Suitable for Rapid Display using Current Graphics Systems

Figure 1: Approximation process.

The continuing popularity of the Phong [10] lighting model[1] is a testament to the importance of including such highlights. Most current graphics API's include a Phong-style lighting model for fast shading. These lighting models are much too simplistic to accurately compute global illumination, but we can still make use of them. Instead of viewing Phong as a lighting model, we can think of it as a set of "appearance basis functions" which can be used to approximately reproduce the results of a more accurate method.

The basic process is outlined in Figure 1. We start from a view-independent non-diffuse global illumination solution. For each non-diffuse object, we fit a set of "virtual lights" that, under the Phong lighting model, will reproduce its computed appearance as closely as possible. By utilizing directionally varying parts of the Phong model, the results will contain non-diffuse aspects of the original solution, although there will also be some loss of directional information due to the limitations of the Phong "basis functions". The results can then be displayed using a standard Phong lighting model.

The translated model can easily be displayed using standard graphics API's (e.g. OpenGL, VRML, or Direct3D) and can even be embedded in display lists. This makes the model portable and suitable for the existing highly optimized 3D graphics display systems. The results are also much more compact than the original global illumination solutions. Most importantly, we apply the lesson of the popular but physically impossible Phong lighting model: even fairly approximate highlights are better than none.

1.1 Related Work

Several researchers have proposed methods for generating and displaying view-independent non-diffuse global illumination solutions (e.g. [6, 11]). Practical application of such methods has so far been hampered by their high computational cost, large storage requirements, and slow display

*[bjw, gun, eric, spf, dpg]@graphics.cornell.edu
http://www.graphics.cornell.edu
1997

[1]In this paper we use the term Phong somewhat loosely to mean the Phong model, the Blinn-Phong model [3] or any similar simple direct lighting model.

speeds. We hope the methods presented here may help push them toward greater use.

Image-based techniques represent a very different route to non-diffuse walkthroughs. They store the illumination in a set of images instead of on surfaces. Image rendering algorithms such as [4, 7, 8] are then used to quickly interpolate new viewpoints from the precomputed images for a walkthrough. These methods offer some potential advantages, but it is not yet known how well they will scale to walkthroughs of larger environments. We consider them to be promising, but take a different approach here.

Environment or reflection maps [1, 5, 12] have long been used for the rapid display of directionally dependent effects. Their main difference from our work lies in their application. They are usually used as a small extension to a simplistic direct lighting model, whereas we are fitting our directional effects in order to reproduce the appearance computed by a physically-based method. In the future, environment maps may be used in a manner similar to our virtual lights.

Multi-pass rendering techniques are another way to perform walkthroughs with non-diffuse effects. They can implement a variety of extensions to the standard Phong lighting model such as shadows, mirror reflections, refraction, and translucency [2]. The results can be striking and they can handle dynamic environments, which is a major advantage. The problem is that the number of passes required per image increases rapidly with the number of lights and the number of lighting effects simulated. To keep the frame rate interactive, one is forced to limit the environment and choose a somewhat *ad hoc* lighting model.

2 OVERVIEW

Before our technique is used, we assume that a view-independent non-diffuse global illumination solution has been computed for the environment of interest. For each object, this solution will specify its appearance as the amount of light leaving (by emission and/or reflection) every point on the object and in each direction. For simplicity we will assume that this information is specified at a number of selected points which we will refer to as vertices.

An example of a directional light pattern leaving a vertex is shown in 2D at the left in Figure 2. Our goal is to reproduce this pattern using parts of the Phong lighting model. The Phong model allows us two kinds of basis functions: a diffuse or directionally invariant type and the “Phong lobes”, or directionally dependent parts, which are caused by specific lights. The diffuse basis is commonly used to encode diffuse global illumination solutions. The new idea of this paper is to also use the “Phong lobes” to approximate non-diffuse appearance as illustrated in Figure 2.

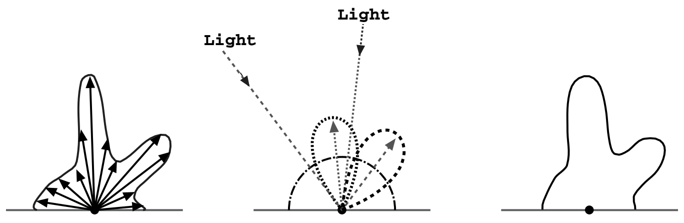


Figure 2: Directional light pattern leaving a vertex. Left: exact or computed pattern, Middle: diffuse basis and two “Phong lobe” basis functions, Right: approximated pattern using the basis functions.

We need to be aware of the many limitations in the Phong model. Some of these make perfect sense (e.g. limit on the number of active lights). Others are somewhat arbitrary and due to the fact that the designers were thinking of Phong as a lighting model rather than as “appearance basis functions”. For instance, there is a specular exponent parameter which controls the width of the Phong lobes. We would like to use different exponents for different lights, and thus fit using lobes of several different sizes. We cannot because in the usual Phong lighting model, the exponent is a property of the surface and not a property of the lights.

Given these various restrictions, we must decide which parts and parameters will be the most useful. For each object, we have chosen to use a single set of directional light sources and a single specular exponent. Additionally, at each vertex we set a diffuse coefficient and a specular coefficient. Together, the exponent, light positions, and light intensities determine the shape of the specular basis function at each vertex as shown in Figure 3. The vertex coefficients then specify the mixture of the diffuse and specular basis functions which will serve as our approximation.

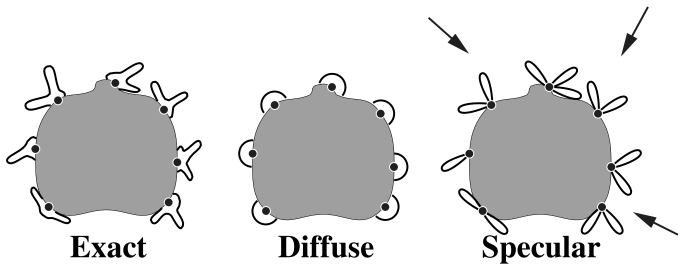


Figure 3: Directional light patterns at selected vertices on an object. Left: exact or computed patterns, Middle: diffuse basis function, Right: specular basis functions induced by three directional lights shown as arrows. Previous methods approximated the exact pattern using only the diffuse basis, while we use both the diffuse and specular.

Setting these parameters is a non-linear optimization problem. At first we tried using a general purpose non-linear optimization procedure, but found that this took a long time and often did not converge. Instead, we have developed a simple set of heuristics for choosing reasonable values. Further optimization could then be done using these values as the initial guess, although we do not currently do this. We iteratively perform a simple three stage fitting process, where a subset of the parameters are set in each stage.

For each object we start by assuming some value for the specular exponent which fixes the shape of the specular lobes, and iteratively fitting a set of lights. We find the brightest value among all vertices and directions on the object, and select the light direction that will create a Phong lobe centered in that direction for that vertex and the light intensity that will reproduce this maximum value (assuming the specular coefficient is 1.0 for now). The effect of this new light is subtracted from each vertex and the process is repeated until some maximum number of lights have been fit.

Once the exponent and lights are chosen, the shape of the specular basis functions is determined. The problem is now a linear optimization, and we set the two coefficients for each vertex using simple least squares fitting. Finally, we repeat this process with different values of the exponent and choose the exponent which gives the best fit in the least squares step.

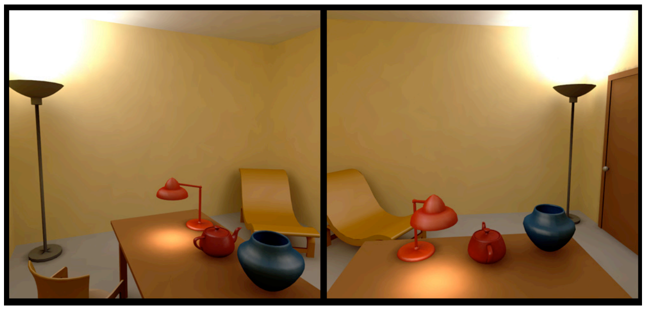


Figure 4: An environment containing a teapot shown using virtual lights.

We cannot expect to achieve an exact fit, but this procedure guarantees there will be highlights in the places where the object has its brightest highlights. Note that each object gets its own set of "virtual lights" which do not affect other objects. These lights do not cast shadows and need not correspond to real lights in the environment. For example, several lights may be used to better simulate a highlight whose shape is different than that of a Phong lobe, or lights may correspond to an indirect light source such as the ceiling above a halogen light.

3 IMPLEMENTATION

For our implementation we have worked with OpenGL's version of the Phong shading model [9].

3.1 OpenGL's Lighting Model

OpenGL uses a simple lighting model to approximate the direct illumination of surfaces by light sources. This lighting model consists of four components: emitted, ambient, diffuse, and specular. These are intended to simulate, respectively: light emitted by a surface (glow), multiply reflected indirect lighting, diffusely reflected direct lighting, and specularly reflected glossy highlights from lights. Lights can be ambient, directional, positional, or spotlights and have ambient, diffuse and specular coefficients. OpenGL guarantees that at least eight lights are available. Surfaces have emitted, ambient, diffuse, and specular coefficients, and a shininess parameter that controls the size of the highlights.

In our implementation, we only use the emitted and specular components, along with directional light sources. The emitted, ambient, and diffuse components all produce directionally invariant lighting at a vertex and are thus redundant for our purposes. We use the emitted component to encode the diffuse part of our solution. The specular component is directionally varying and depends on the shininess of the material, the light direction relative to the surface, the surface normal, and the viewing direction relative to the surface.

Using only the emitted and specular components, the OpenGL lighting equation for determining vertex colors becomes:

$$\text{emitted} + \sum_{\text{lights}} \max(\mathbf{s}\cdot\mathbf{n}, 0)^{\text{shininess}} * \text{specular}_{\text{light}} * \text{specular}_{\text{vertex}}$$

where $\mathbf{n}$ is the vertex normal, and $\mathbf{s}$ is the vector obtained by adding the light direction and the view direction and normalizing.

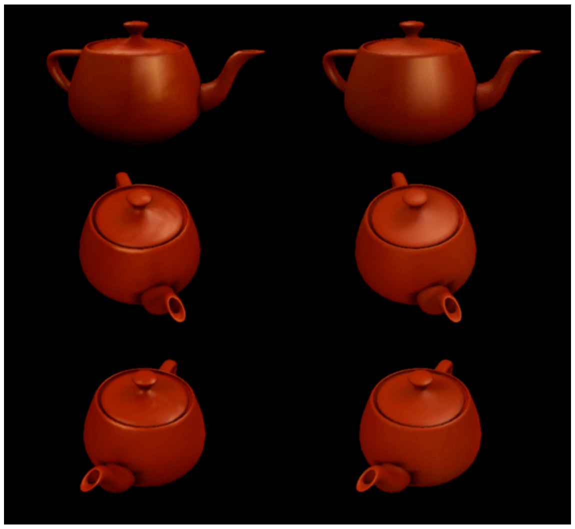


Figure 5: Comparison of original data for the teapot (left) and our approximation using 8 lights (right) shown from three viewpoints.

3.2 The Fitting Process

We compute the initial data by computing a radiosity solution via density estimation [13] and then performing a gather at each vertex and storing the results for a discrete set of outgoing directions. The details are not important and many other methods are possible. For each object, we then need to find the emitted and specular values for each vertex, the directions and intensity values for its lights, and its shininess value. Our algorithm for a single object is:

```
Repeat
  Choose a shininess value
  Repeat
    Subtract effects of existing lights from input light data
    Find the maximum difference
    Add directional light to cause a highlight at this maximum
    Set light intensities to match input data at their maxima
  until all lights have been fit
  For each vertex
    Set emitted and specular values by least squares fitting
until shininess search is done
```

We search for the shininess (i.e. exponent) which minimizes the least squares error. We currently use a golden section search method which eliminates a portion of the search interval on each iteration.

4 RESULTS

We computed a global illumination solution for the environment shown in Figure 4 displayed using eight "virtual lights" per specular object. This scene contains the familiar Utah teapot which we use as a an example object to demonstrate our results. A comparison between the computed teapot and our fitted approximation with eight "virtual lights" is shown in Figure 5. The results are perceptually convincing overall although small differences can easily be seen. We also compare results when using fewer fitted lights in Figure 6.

The real test of our techniques is in walkthroughs of non-diffuse environments. We can only show images here, but

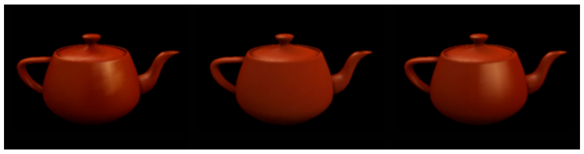


Figure 6: Comparison of original data for the teapot (left) and our approximation using 2 (middle) and 8 (right) virtual lights.

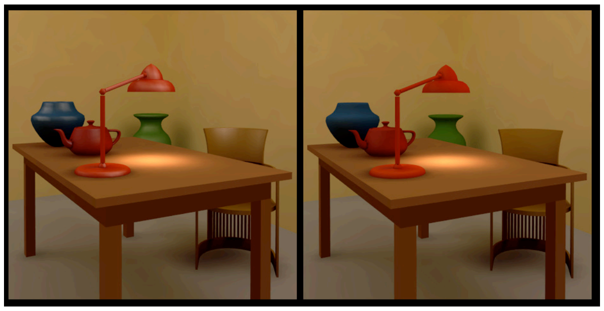


Figure 7: Our environment shown with virtual lights and diffuse only (virtual lights turned off).

we have included a live walkthrough of our environment in the video proceedings. Figure 7 shows images of this environment both with and without the virtual lights to demonstrate how much they contribute perceptually.

4.1 Limitations and Open Issues

While our results match the computed solutions surprisingly well, there are many limitations to how well we can currently mimic real appearance. Some of these are fundamental to the technique (e.g. mirrors simply require too much directional information), but others could be alleviated with changes in both our implementation and in the graphics display interfaces.

Some improvements, such as using positional instead of directional lights or finding a perceptually better fitting process, are possible now. But many others would require extensions or additions to the current graphics API's. Some potentially useful extensions would be the ability to vary the specular exponent per light and having a separate specular coefficient for each light at each vertex.

Gouraud interpolation is a major source of artifacts and requires that the curved surfaces be finely tessellated. True Phong shading would reduce these problems, but is rarely available because it is more computationally demanding.

While we use standard graphics API's, we use them in a hitherto unusual way. Many systems are not properly optimized for the sequence of operations we use. For instance on many OpenGL systems there is a very large ($> 4\times$) performance penalty for varying more than one property per vertex (in our case emitted and specular coefficients). We have achieved good performance using a two pass technique, one pass for the diffuse component and a second for the specular. Another possibility is to leave the specular coefficient fixed at the cost of some additional loss of quality. But this problem should largely disappear if our method gains acceptance and is considered during system optimization.

5 CONCLUSIONS

We have presented a technique for using a simple shading model, such as Phong, to approximate the non-diffuse appearance calculated by some more accurate method. This technique can translate view-independent non-diffuse global illumination solutions into a form that is more compact, portable, and suitable for fast display. This allows for non-diffuse walkthroughs which are perceptually better than traditional diffuse-only walkthroughs.

Moreover, by targeting our results toward standard graphics API's such as OpenGL, we can utilize the existing 3D graphics display infrastructure and allow designers to easily optimize their systems for our type of solutions. Finally, we have suggested a few ways in which future graphics API's could be enhanced to better enable the reproduction of non-diffuse appearance.

Acknowledgments

Our thanks to Ben Trumbore, the modelers, and our anonymous reviewers. This work was supported by the NSF Science and Technology Center for Computer Graphics and Scientific Visualization (ASC-8920219) and by NSF ASC-9523483 and performed with workstations generously donated by the Hewlett-Packard Corporation.

References

[1] J. Blinn and M. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, October 1976.

[2] P. J. Diefenbach. *Pipeline Rendering: Interaction and Realism through Hardware-base Multi-pass Rendering.* PhD thesis, University of Pennsylvania, 1996.

[3] A. S. Glassner. *Principles of Digital Image Synthesis.* Morgan-Kaufman, San Francisco, 1995.

[4] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. Cohen. The lumigraph. *Computer Graphics*, pages 43–54, August 1996. ACM Siggraph '96 Conference Proceedings.

[5] N. Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, Nov. 1986.

[6] D. S. Immel, M. F. Cohen, and D. P. Greenberg. A radiosity method for non-diffuse environments. *Computer Graphics*, 20(4):133–142, August 1986. ACM Siggraph '86 Conference Proceedings.

[7] M. Levoy and P. Hanrahan. Light field rendering. *Computer Graphics*, pages 31–42, August 1996. ACM Siggraph '96 Conference Proceedings.

[8] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. *Computer Graphics*, pages 39–46, August 1995. ACM Siggraph '95 Conference Proceedings.

[9] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide.* Addison-Wesley, New York, 1993.

[10] B.-T. Phong. Illumination for computer generated images. *Communications of the ACM*, 18(6):311–317, June 1975.

[11] F. X. Sillion, J. Arvo, S. Westin, and D. Greenberg. A global illumination algorithm for general reflection distributions. *Computer Graphics*, 25(4):187–196, July 1991. ACM Siggraph '91 Conference Proceedings.

[12] D. Voorhies and J. Foran. Reflection vector shading hardware. *Computer Graphics*, 28(3):163–166, July 1994. ACM Siggraph '94 Conference Proceedings.

[13] B. Walter, P. M. Hubbard, P. Shirley, and D. P. Greenberg. Global illumination using local linear density estimation. *ACM Transactions on Graphics*, October 1997.

Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware

Wolfgang Heidrich and Hans-Peter Seidel

Computer Graphics Group, University of Erlangen
Am Weichselgarten 9, D-91058 Erlangen, Germany
Email: heidrich@informatik.uni-erlangen.de

Abstract

There are surprisingly many anisotropically reflecting objects in the real world. Some examples are brushed metal, cloth, CDs, and even many brands of paper.

Despite the importance of anisotropic reflections in the real world, most of todays computer graphics systems cannot deal with them appropriately. This is particularly true for interactive and hardware-accelerated rendering systems, which is mostly due to the fact that so far there has been no model for simulating anisotropy with commonly available computer graphics hardware.

In this paper we describe a novel approach for simulating anisotropic reflections on OpenGL-based architectures in real time. The rendering of an anisotropic surface with one light source is exactly as expensive as the rendering of the same surface with traditional texturing and lighting. It is therefore possible in real time, even on low-end workstations and PCs.

1 Introduction

Many of the objects we know from day-to-day life have anisotropic surfaces. Often, this anisotropy is caused by a micro structure, in which long, thin features are aligned in one predominant direction. For example, brushed metal objects have been polished in such a way that there are a lot of parallel scratches in the surface. On CDs and records, the micro structure is given by the tracks carrying the music or data. Woven cloth consists of fibers oriented in certain directions, and paper contains wooden grains, which, due to the production process, are often oriented mostly in one direction [8].

The anisotropy in all these examples is created because the distribution of the surface normals along the scratches or fibers is different from the distribution across them. If the observer is distant enough from the surface, these features cannot be seen individually. Instead, the impression of a homogeneous surface surface is created (see Figure 1).

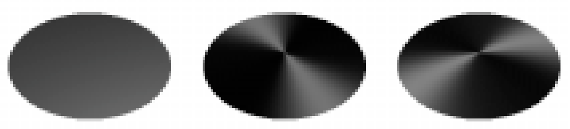


Figure 1: An example of a disk shaded with isotropic reflection (left), anisotropic reflection with circular features (center), and radial features (right).

A model for this kind of surface has been described in [1]. The scratches of fibers are assumed to be lines or curves on the surface. With this assumption, existing lighting models for the illumination of lines can be applied to rendering anisotropic surfaces.

The remainder of this paper is organized as follows. In Section 2 we briefly summarize the method for illuminating lines described in [1]. In Section 3 we describe a way to efficiently illuminate lines with the help of computer graphics hardware and the OpenGL API [5]. This part is based on work presented in [11] and [9]. Afterwards we discuss how self-shadowing of surfaces can be handled, and present multipass methods for isotropic components and multiple light sources in Section 4.

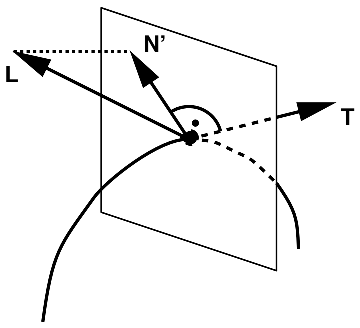


Figure 2: In order to find the shading normal N, the light vector L is projected into the normal plane.

2 Illumination

The illumination of lines and line segments in 3-dimensional space has been discussed in several places [1, 11, 9]. Other publications (for example [3, 2]) have researched the same topic in the context of rendering hair and fur.

Before we discuss the illumination of lines, we briefly review the Phong illumination model of surfaces. The intensity I_o of any surface point is given as the sum of the ambient, diffuse and specular components:

$$\begin{aligned} I_o =& I_{ambient} + I_{diffuse} + I_{specular} \\ =& k_a I_a + (k_d \cos\phi + k_s(\cos\theta)^n) \cdot I_i \\ =& k_a I_a + (k_d \langle L, N\rangle + k_s \langle V, R\rangle^n) \cdot I_i \end{aligned}$$

where k_a, k_d, and k_s are the ambient, diffuse, and specular reflection coefficients, $1/n$ is the surface roughness, I_a is the ambient light in the scene, I_i is the incoming light at the surface point, N is the surface normal, L is the unit vector towards the light, R is the reflection of L at N, and finally V is the unit vector towards the viewpoint.

The fundamental difference between the illumination of curves and the illumination of surfaces is that every point on a curve has an infinite number of normal vectors. Thus, every vector that is perpendicular to the tangent vector of the curve is a potential candidate for use in the illumination calculation.

For reasons described in [1], and [9], the vector N' selected from this multitude of potential normal vectors is the projection of the light vector L into the normal plane, as depicted in Figure 2.

As described in [11, 9], the cosine $\langle L, N'\rangle$ used for the diffuse component can be expressed in terms of the tangent T, and the light vector V for this specific normal vector N'. Since L, T, and N' are all coplanar, the angle $\phi = \angle(L, N')$ is identical to $\pi/2 - \angle(N', T)$, and thus

$$\langle L, N'\rangle = \cos\phi = \sin(\angle(N', T)) = \sqrt{1 - \langle L, T\rangle^2}.$$

Similarly, for the specular component we get

$$\langle V, R\rangle = \sqrt{1-\langle L, T\rangle^2}\sqrt{1-\langle V, T\rangle^2} - \langle L, T\rangle \langle V, T\rangle .$$

With this transformation we can describe the brightness of a point as a bivariate function $I_o(\langle L, T\rangle, \langle V, T\rangle)$, which only depends on the two cosines $\langle L, T\rangle$ and $\langle V, T\rangle$.

With the exception of self-shadowing effects, which are described in Section 3.1, this method for illuminating lines is directly applicable for the illumination of anisotropic surfaces as described above.

3 OpenGL Implementation

For an efficient implementation of this model in OpenGL, let us assume we had a way of specifying $\langle L, T\rangle$ and $\langle V, T\rangle$ as the *texture coordinates* s and t (actually, in order to make for valid OpenGL texture coordinates, the cosines first have to be mapped to the range $[0, 1]$). In order to illuminate any given point of a curve or surface with this illumination model, we could then simply precompute a texture containing the bivariate function $I_o(\langle L, T\rangle, \langle V, T\rangle)$.

One way of using $1/2(\langle L, T\rangle + 1)$ and $1/2(\langle V, T\rangle + 1)$ as texture coordinates, is to recompute and specify these values for each frame individually. However, this would mean that the geometry of the scene would have to change for each frame, which would result in low performance. A more clever way of achieving the same goal is to use a texture coordinate matrix for computing the cosines [9]. If we use the tangent vector $T = (t_x, t_y, t_z)^T$ as the initial texture coordinate vector, then an appropriately chosen texture matrix performs the necessary transformations:

$$\frac{1}{2}\begin{bmatrix} l_x & l_y & l_z & 1 \\ v_x & v_y & v_z & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} t_x \\ t_y \\ t_z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2}(\langle L, T\rangle + 1) \\ \frac{1}{2}(\langle V, T\rangle + 1) \\ 0 \\ 1 \end{bmatrix}$$

Here, the l_i and v_i are the components of the L, and V vector, respectively.

Note that this allows us to specify the texture coordinates only once, while changes in view point and light position can be taken care of by manipulating the texture matrix. As a consequence, the whole geometry remains fixed for all frames, and can be stored in a display list. Thus, anisotropic surfaces can be rendered at exactly the cost of rendering the same surface with traditional texture mapping enabled.

Figure 3 shows a sphere rendered with this method from different viewpoints. The sphere resembles a satin ball like the ones used for Christmas trees. The satin fibers range from pole to pole on the sphere. The light comes from a 45 degree angle from the right and from behind the viewer.

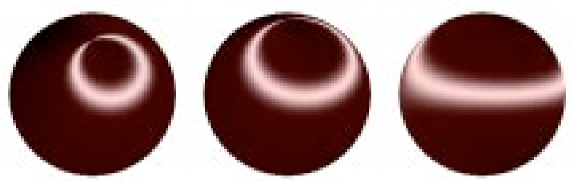


Figure 3: A "satin ball" rendered with the proposed method from three different views.

3.1 Self-Shadowing

Although this image already shows the typical effects of anisotropy, the shading looks unrealistic, because the left side of the sphere is illuminated despite it pointing away from the light source. This behavior is correct for the illumination of lines, because these do not have back faces.

For surfaces, however, areas that point away from the light should not receive any light. Moreover, areas that are illuminated under glancing angles should be darker than areas onto which the light shines perpendicularly. This is due to the fact that, for glancing angles, the fibers (or scratches) cast shadows onto each other. In order to account for this effect, Banks [1] proposes to modify the illumination model as follows:

$$I_o = I_{ambient} + clamp(\langle -N, L\rangle\ (I_{diffuse} + I_{specular}),$$

where N is the actual geometric normal at the surface point, and the function *clamp* is zero for negative values, and the identity function otherwise.

This modification can be incorporated into the OpenGL implementation with the help of the standard OpenGL lighting mechanism. In the OpenGL rendering pipeline, the result of the texture lookup operation for each pixel can be multiplied by the result of the lighting calculation ("fragment color") for the same pixel. If we use a diffuse, white surface material, and a white, parallel light, the resulting lighting computations performed by OpenGL result exactly in the clamped cosine value required above. Figure 4 shows some results for this modified algorithm.

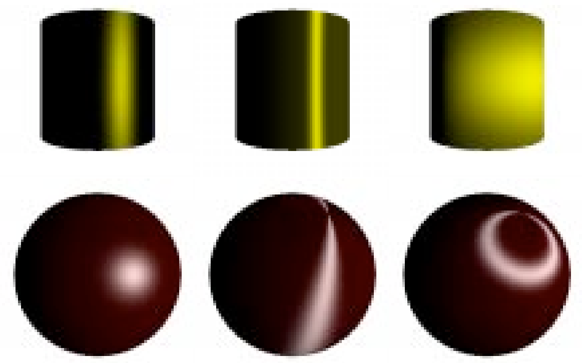


Figure 4: Sphere and cylinder with isotropic reflection (left), anisotropic reflection with radial features (center), and vertical features (right).

This modified method slightly increases the cost of the rendering algorithm. Now, not only texturing, but also lighting has to be performed by OpenGL.

4 Multipass Techniques

So far, we have only considered the illumination through a single light source. If we add a second light source, say in direction L', the illumination becomes a function in three variables: $I_o\left(\langle L,T\rangle, \langle L',T\rangle, \langle V,T\rangle\right)$, which could be represented as a 3-dimensional texture map. 3-dimensional texturing is available as an extension on many OpenGL platforms today, and is proposed to become a standard feature of OpenGL version 1.2 [6]. The modified texture matrix for two light sources is as follows:

$$\frac{1}{2}\begin{bmatrix} l_x & l_y & l_z & 1 \\ l'_x & l'_y & l'_z & 1 \\ v_x & v_y & v_z & 1 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Unfortunately, 3-dimensional textures can grow quite large, which, in turn, causes performance problems. Also, this method fails for more than two light sources. A better way to illuminate anisotropic surfaces with more than one light source is to use a multipass approach.

For this multipass method, the object is rendered multiple times, once for each light source. The results of each rendering pass are added to the image using alpha blending. Obviously, the rendering time increases linearly with the number of light sources. Nonetheless, for a relatively small number of lights, or a small percentage of anisotropic surfaces, the method is still fast, even on low end machines.

Another application of multipass techniques is the addition of isotropic reflection components. The model described so far is purely anisotropic. Many surfaces, however, have both an isotropic and an anisotropic reflection component, for example when the scratches or fibers on the surface have a certain, non-zero distance from each other. A single rendering pass with the standard OpenGL lighting model can add this isotropic component for all the light sources in the scene.

Other surfaces have anisotropies in multiple directions. For example woven cloth consists of fibers oriented in two perpendicular directions. Again it is possible to use multiple passes, one for each direction and light source, to account for these effects.

5 Conclusion

In this paper we have presented a novel approach for rendering surfaces with anisotropic reflection characteristics in OpenGL. The method requires one rendering pass per light source, and each pass is no more expensive than rendering the same object with traditional lighting and texture mapping switched on. Therefore, the method is well suited even for PCs and low end workstations, as long as the number of light sources is not too large.

References

[1] David C. Banks. Illumination in diverse codimensions. In *Computer Graphics (Proceedings of SIGGRAPH '94)*, pages 327–334, July 1994.

[2] Agnes Daldegan, Nadia Magnenat Thalmann, Tsuneya Kurihara, and Daniel Thalmann. An integrated system for modeling, animating and rendering hair. In *Eurographics '93*, pages 211–221, 1993.

[3] Ken ichi Anjyo, Yoshiaki Usami, and Tsuneya Kurihara. A simple method for extracting the natural beauty of hair. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 111–120, July 1992.

[4] James T. Kajiya. Anisotropic reflection models. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, pages 15–21, July 1985.

[5] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison Wesley, 1993.

[6] OpenGL ARB. *OpenGL Specification, Draft Version 1.2*, 1997.

[7] Pierre Poulin and Alain Fournier. A model for anisotropic reflection. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 273–282, August 1990.

[8] Morgan Schramm, Jay Gondek, and Gary Meyer. Light scattering simulations using complex surface models. In *Graphics Interface '97*, pages 56–67, 1997.

[9] Detlev Stalling, Malte Zöckler, and Hans-Christian Hege. Fast display of illuminated field lines. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):118–128, 1997.

[10] Gregory J. Ward. Measuring and modeling anisotropic reflection. *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 265–273, July 1992.

[11] Malte Zöckler, Detlev Stalling, and Hans-Christian Hege. Interactive visualization of 3D-vector fields using illuminated stream lines. In *IEEE Visualization '96*, pages 107–113, 1996.

Interactive Reflections on Curved Objects

Eyal Ofek Ari Rappoport

Institute of Computer Science, The Hebrew University

Abstract

Global view-dependent illumination phenomena, in particular reflections, greatly enhance the realism of computer-generated imagery. Current interactive rendering methods do not provide satisfactory support for reflections on curved objects.

In this paper we present a novel method for interactive computation of reflections on curved objects. We transform potentially reflected scene objects according to reflectors, to generate *virtual objects*. These are rendered by the graphics system as ordinary objects, creating a reflection image that is blended with the primary image. Virtual objects are created by tessellating scene objects and computing a virtual vertex for each resulting scene vertex. Virtual vertices are computed using a novel space subdivision, the *reflection subdivision*. For general polygonal mesh reflectors, we present an associated approximate acceleration scheme, the *explosion map*. For specific types of objects (e.g., linear extrusions of planar curves) the reflection subdivision can be reduced to a 2-D one that is utilized more accurately and efficiently.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism.

Keywords: ray tracing, interactive reflections, virtual objects method, reflection subdivision, explosion map.

1 Introduction

Interactive photo-realistic rendering is a major goal of computer graphics. Global view-dependent illumination phenomena greatly enhance image quality. An extremely important type of view-dependent phenomenon is reflection. Reflections on curved object are not supported well by current interactive rendering techniques. In this paper we address the problem of interactive rendering of reflections on curved objects.

Background. Current interactive graphics systems utilize hardware acceleration that directly supports hidden surfaces removal, simple local shading models and texture mapping. While the polygon throughput of these systems is impressive, the range of shading effects they provide hasn't changed much since their introduction. In particular, they lack support for global illumination phenomena in dynamic scenes.

Global illumination phenomena greatly enhance the quality of synthetic imagery. They can be coarsely classified to view-independent and view-dependent phenomena. Among the former, diffuse illumination in static scenes [Sillion89] and shadows [Segal92] can be interactively rendered using current hardware. However, global *view-dependent* phenomena are crucial for providing life-like realism. When only view-independent effects are provided, the visual nature of the result can be dull and lifeless, even when the scene is dynamic.

An extremely important view-dependent illumination phenomenon is reflection. The dominant method for generating reflections is ray tracing [Whitted80, Glassner89]. In spite of extensive work on ray tracing acceleration schemes, [Jansen93] states that the only hope for interactive ray tracing lies in massively parallel computers, and even then satisfactory performance is not guaranteed.

Environment mapping [Blinn76, Greene86, Haeberli93, Voorhies94] generates at interactive rates reflections that are approximately correct when the reflected objects are relatively far from the reflector. However, when this condition is violated the results are of very poor accuracy.

It is well-known that reflections on planar surfaces can be generated by (1) mirroring the viewer along the reflecting plane, (2) creating a reflection image by rendering the scene from the new point of view, and (3) merging the main image with the visible portion of the reflector in the reflection image. Surprisingly, although this method can significantly accelerate ray tracing, it has been accurately documented only recently. The descriptions in [Foley90] (in which the method is called 'reflection mapping') and [McReynolds96] are correct only when the original viewer and all objects lie on the same side of the reflecting plane. A correct description is given in [Hall96]. [Diefenbach97] shows how to use variants of this method for interactive simulation of various general reflectance functions of planar objects. The concept of a reflected virtual world was also used in [Rushmeier86, Wallace87, Sillion89] for supporting specular reflections from planar objects in a radiosity context.

Contribution. In this paper we present a method for interactive rendering of reflections on *curved* objects, based on merging a primary image and a reflection image. The reflection image is generated by creating and rendering *virtual objects* corresponding to reflections of scene objects. Virtual objects are rendered like ordinary polygons, thus taking advantage of the features supported by the graphics system. They are created using a structure called the *reflection subdivision* and an associated approximate acceleration scheme, the *explosion map.*

The method presents a novel approach to the computation of reflections in computer graphics, and is unique in providing approximate reflections on curved objects at interactive rates. Moreover, the rendered scenes can be completely dynamic; no pre-processing is necessary. The method provides higher quality than environment mapping, because it allows reflected objects to be nearby the reflector and it supports equally well reflectors having a large curvature. For scenes in which reflected images of objects occupy more than a

Institute of Computer Science, The Hebrew University, Jerusalem 91904, Israel. http://www.cs.huji.ac.il/∼arir,∼eyalp arir,eyalp@cs.huji.ac.il

few pixels and in which the depth complexity of the reflection image is not large, the method is much more efficient than ray tracing, because it efficiently exploits the spatial coherency of the reflection image. The price paid for the advantages of the method is that its performance is less efficient than that of environment mapping and the generated images are only polygonal approximations (as in most interactive systems). In addition, its accuracy depends upon the geometric nature of the reflector.

The paper is structured as follows. Section 2 gives an overview of the method. Sections 3, 4 and 5 deal with convex reflectors, discussing respectively the reflection subdivision, the explosion map, and special reflectors. Section 6 deals with non-convex reflectors. Results and an in-depth discussion are given in Sections 7 and 8.

2 Method Overview

In this section we give an overview of the virtual objects method. We present the general idea (2.1), image merging alternatives (2.2), a brief discussion on planar reflectors (2.3), and a high-level outline on non-planar reflectors (2.4).

2.1 General Idea

The virtual objects method is inspired by the following observation. Consider an image containing reflections. Two kinds of entities are visible: reflecting objects, or *reflectors*, and *reflected images* of reflected 3-D objects. When the reflector is a perfect planar one, the geometry of the reflected images is identical to images of the reflected objects from some other viewpoint. In fact, we cannot distinguish between 'real' objects and reflected images of objects. Interior designers utilize this phenomenon when covering walls with mirrors in order to make rooms seem larger. For non-planar reflectors, the appearance of reflected objects is a deformed version of their ordinary appearance. In general, there is no viewpoint from which they appear identical to their reflected images. The nature of the deformation depends upon the geometry of the reflector. Convex reflectors deform reflected objects to seem smaller, and concave reflectors produce reflected images that may seem larger than the reflected object or degenerate into strange chaotic images.

This observation inspires the following algorithm for generation of reflections (Figure 1): for every reflector and every object potentially reflected in it, compute a *3-D virtual object,* that, when rendered using ordinary 3-D rendering methods, will produce an image having a visual appearance similar to the object's reflected image. If depth relationships between the virtual objects are still correct, the rendered images of the virtual objects can be merged together using some hidden surfaces removal algorithm. The result can now be alpha blended with a reflector image containing view-independent lighting to produce the final image. The alpha blending coefficients are determined by the relative reflectivity of the reflector.

```
SceneRender (Scene $S$, View $E$):
(1) Render $S$ without reflections into primary image $I$.
(2) For every visible reflector $R \in S$
(3)     For every potentially reflected object $O \in S$
(4)         $O' \leftarrow$ VirtualObject ($R, O, E$).
(5)         Render $O'$ into a reflection image $I'$.
(6)         If multiple levels of reflections are desired
                Call the algorithm recursively.
(7)     Alpha blend $I'$ and $I$, according to
        the reflectivity of $R$.
```

Figure 1 The virtual objects method.

When virtual objects can be computed efficiently, the resulting method is very attractive, since reflected images are generated at the object, rather than the pixel, level. Most of this paper deals with step 4, efficient generation of virtual objects. Naturally, only visible reflectors are considered, and the scene can be stored in a data structure that supports culling of scene objects that cannot be reflected.

A comment about shading: for planar reflectors we can reflect the light sources as well as the scene objects and simply use the reflected ones. For non-planar reflectors, it is more accurate to compute shading values for vertices at the world coordinate system, and then use these values for the virtual vertices. On current architectures, this shading is most efficiently computed in software, and the hardware is used for rasterization and texturing.

2.2 Image Merging

The primary and reflection images can be merged in two ways. First, the reflection image can be used as a texture when rendering a reflector. Alternatively, the reflection image can be directly rendered on the screen (using a stencil bit-plane defining the screen image of the reflector.) The view-independent component of the reflector is now rendered, alpha blending it with the reflection image.

Texture mapping and stencil-guided image merging are standard features in interactive graphics systems, even current low-end ones. The choice of method depends on the actual graphics architecture available, especially on its memory organization. For more details, see [Ofek98, McReynolds96, Hall96].

2.3 Planar Reflectors

The method of [McReynolds96, Hall96, Diefenbach97] is a special case of the virtual objects method, when the reflectors are planar and when we consider the objects, rather than the viewpoint, as being mirrored. Note that in this case the method is essentially an image-space version of beam tracing [Heckbert84]. An attractive property of planar reflectors is that the location of a virtual point is a simple affine transformation, *mirroring,* of the real point. Moreover, this transformation does not depend on the viewer location, only on that of the reflector. In Figure 2(a), the location of the virtual image Q' of a scene point Q remains constant for two viewpoints E_1 and E_2. Hence, the same simple affine transformation can be used for all reflected polygons. Full details on how to generate the mirroring transformation are given in the above references.

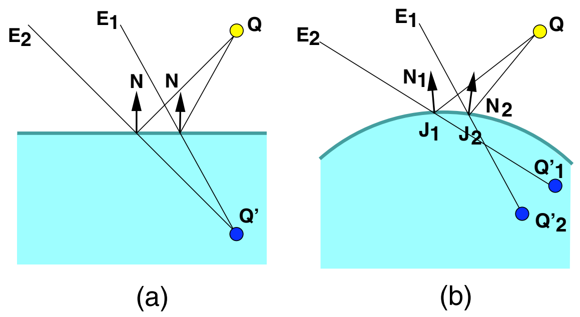


Figure 2 For planar reflectors, the virtual location of a point does not depend upon the viewpoint (a). This does not hold for curved reflectors (b).

Note that as presented so far, the method produces correct results only when the viewpoint and the reflected polygon are on the same side of the reflector. Consider a polygon lying on the other side of the reflector. After the mirroring transformation, it can erroneously obscure the reflector from the viewer, because they lie on the same side of it. This problem can be overcome by not mirroring a polygon if all of its vertices lie behind the reflector. The test is done by plugging the vertex coordinates into the reflector plane equation and testing the sign of the result. However, this method does not solve the case when the polygon lies only partially behind the reflector. In many cases such polygons do not cause incorrect results because the virtual front part falls outside of the reflector stencil anyway. For planar reflectors, the problem can be solved very efficiently by defining the reflector plane as a front clipping plane.

2.4 Non-Planar Reflectors

Generation of virtual objects for non-planar reflectors is more difficult than for planar reflectors, because the main property of the planar case does not hold: the location of a virtual point is not a simple affine transformation independent of the viewer position (Figure 2(b)). In general, every reflected point is transformed differently.

Our approach is outlined in Figure 3. The reflected object is tessellated into polygons (step 1). The fineness of the tessellation depends upon the desired accuracy of the resulting reflection image. Tessellations are further discussed in Sections 7 and 8. In steps 2–5, virtual polygons are generated by computing virtual vertices for the tessellation vertices. The collection of all virtual polygons forms the desired virtual object rendered in step 5 of Figure 1. The main step is 4, computing a single virtual vertex; its description occupies much of the rest of the paper.

```
VirtualObject (Reflector $R$, Object $O$, View $E$):
(1) Tessellate $O$ into polygons.
(2) For each polygon $P$
(3)      For each vertex $Q$ of $P$
(4)           $Q' \leftarrow$ VirtualVertex $(R, Q, E)$.
(5)      Connect the $Q'$s to form a virtual polygon $P'$.
(6) Connect the $P'$s to form the virtual object $O'$.
```

Figure 3 Computing a virtual object O' for a potentially reflected object O on a non-planar reflector R.

Rendered polygons are consistent and possess no holes, because virtual objects are formed by connecting virtual vertices. Visibility relationships between virtual objects are preserved due to the usage of a hidden surfaces removal mechanism (in practice, a z-buffer) for them.

3 The Reflection Subdivision

In this section we start detailing our approach towards computing virtual vertices for curved reflectors. We assume here that the reflector is convex. Concave and other non-convex reflectors are discussed in Section 6. Our approach is based on approximating the reflector by a polygonal mesh. In many cases this is the format in which objects are given anyway; when they are given in a higher-level representation (e.g., a NURBS surface) they are tessellated. For simplicity, we assume that mesh polygons are triangles, but this is not necessary.

Intuition. Given a reflector R and an arbitrary scene point Q, we want to generate the corresponding virtual point Q' (consult Figure 2(b)). If we knew the point of reflection J and normal N on the boundary surface of R, we could easily compute Q' by mirroring Q along the tangent plane to R at J. In some cases, when we know the geometric nature of the reflector (e.g., a sphere), J can be computed by a direct formula. However, for a general convex polygonal mesh there is no direct formula.

We use an approximation. Every reflector triangle defines two space cells: a *reflected cell* and a *hidden cell.* Suppose that we can find the cell C, defined by triangle T, in which the scene point Q lies. A naive method would mirror Q across the plane containing T. However, this would clearly show the linear approximation of the reflector (imagine a reflecting sharply cut diamond!). Instead, we use the relative location of Q inside C to define a triplet of barycentric coefficients. These coefficients are used to interpolate the three tangent planes at the vertices of T, yielding a new tangent plane that is now used for mirroring Q.

In this section we study the space subdivision defined by the reflector and also explain why we need to compute virtual points for points that are not reflected. The full details of the computation are given in Sections 4 and 5.

The subdivision. Each vertex V_i of the tessellated reflector possesses a normal N_i. Reflector vertices are either *front-facing* or *back-facing,* according to whether their normals point towards or away from the viewer (a normal orthogonal to the line of sight is considered front-facing). Due to the convexity of the reflector, every front-facing vertex is visible by the viewer (when there are no other obscuring objects). Note that back-facing vertices might still be visible (this is a tessellation artifact). When all vertices of a mesh triangle are front-facing (back-facing), we refer to the triangle as being front-facing (back-facing). Otherwise we say that the triangle is a *profile* triangle.

For each front-facing vertex V_i we define two rays: (1) a *reflection ray* R_i, mirroring the ray from V_i to the viewer across the normal N_i, and (2) a *hidden ray* H_i, originating at V_i and extending to infinity in the opposite direction to that of the viewer. Figure 4 shows a 2-D version of the situation. In (a), reflection rays are shown in red and hidden rays in blue.

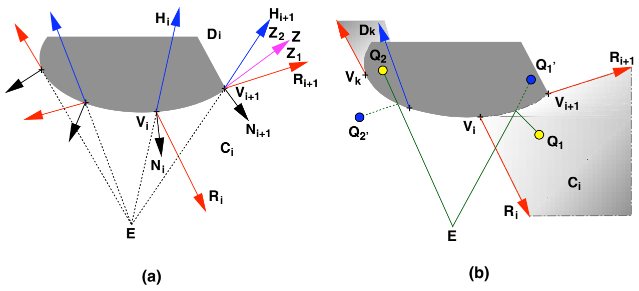


Figure 4 (a) The reflection subdivision in 2-D. C_i and D_i are the reflected and hidden cells defined by reflector vertices V_i, V_{i+1}. The ray Z bisects the unreflected region on the right into two parts Z_1, Z_2. (b) Computation of virtual vertices: the point Q_1 in the reflected cell C_i is transformed to Q_1' inside the hidden cell D_i; the point Q_2 in the hidden cell D_k is transformed to Q_2' outside the reflector in the reflected cell C_k.

Two reflection rays R_i, R_j corresponding to adjacent front-facing mesh vertices V_i, V_j define a ruled bi-linear parametric surface $s(V_i + tR_i) + (1 - s)(V_j + tR_j)$. Note that in general this surface is not planar, because the two rays are usually not co-planar. The two hidden rays H_i, H_j span an infinite truncated triangle containing the edge V_i, V_j.

Now consider the three vertices V_i, V_j, V_k of a front-facing mesh triangle V_{ijk}. The triangle induces two space regions: (1) A *reflected cell* C_{ijk} bounded by the three ruled surfaces corresponding to the triangle edges and by V_{ijk} itself (figure 10). (2) A *hidden cell* D_{ijk}, which is the infinite part of the truncated pyramid bounded by V_{ijk} and the triangles spanned by the hidden rays. We refer to the union of the reflected (hidden) cells as the reflected (hidden) region.

An important property of the reflected and hidden cells is that they do not intersect each other, since the reflector is convex. Therefore, we can define the *reflection subdivision* as the subdivision of space induced by these cells. Note, however, that these cells do not *cover* space; we call the part of space not covered by reflection or hidden cells the *unreflected region*. In Figure 4(a), the part of the unreflected region lying on the right side of the reflector is the union of Z_1, Z_2 (the reason for subdividing this region and the meaning of the ray Z are explained below). Points in the unreflected region can (in principle) be seen by the viewer, but cannot be reflected by the reflector. A point is potentially reflected by the reflector if and only if it lies in the reflected region. We say 'potentially' because its reflection may be obscured by the reflection of another point.

The unreflected and hidden regions. We compute virtual images for vertices of potentially reflected scene polygons (Figure 3, step 4). These virtual vertices are connected in order to generate virtual polygons, which are then rendered to create the reflection image (Figure 1, step 5). Scene polygons that lie completely in the hidden or unreflected regions can be discarded. However, *mixed polygons*, lying partially in these regions and partially in the reflected region, pose a problem. For such polygons, we would like to render the reflection of the part that lies in the reflected region. However, if we compute only one or two virtual vertices, we would not be able to connect these in order to generate virtual polygons. In some sense, the vertices lying in hidden or unreflected regions are representatives of a polygon area that we want to see reflected.

A naive way to deal with mixed polygons is to intersect them (exactly or approximately) with the region boundaries, thus forcing them to have a uniform classification. However, this is inefficient because the regions depend on the viewpoint. Another way is to subdivide them into smaller polygons, effectively doing an adaptive tessellation of scene objects. Subdivision is stopped when the 'lost' areas are deemed to be small enough.

A more efficient and elegant method is to define a virtual vertex for *every* polygon vertex, even for hidden and unreflected ones (e.g., vertex Q_2 in Figure 4(b)). These *doubly virtual* vertices are not real reflections; their sole purpose is to 'close' virtual polygons so that the graphics system could render them. In general, they lie outside the image of the reflector. Note that this actually is the approach taken in the planar reflector case. Hidden cells are easy to take care of, because there is a one-to-one correspondence between hidden and reflected cells. Moreover, it is possible to define a transformation that maps a reflected cell to exactly cover the corresponding hidden cell, and maps a hidden cell to exactly cover its corresponding reflected cell (see Section 4.2).

The unreflected region is more problematic. We would like to define a transformation for this region such that (1) the part of the region adjacent to a reflected cell will be transformed to be adjacent to its corresponding hidden cell (and vice versa), and (2) there is some continuity of the transformation between the unreflected and the reflected regions. To achieve such a transformation, we define for every contour edge of the reflector an auxiliary *bisecting surface* Z, which extends the edge into the unreflected region. Figure 4(a) shows a 2-D example. In 2-D we have a contour vertex and not a contour edge (it is simply the extreme vertex V_{i+1}), and the bisecting surface is simply a ray Z. Z is orthogonal to the normal N_{i+1} at V_{i+1} and extends V_{i+1} into the unreflected region, thus bisecting the region into two parts Z_1, Z_2. The desired transformation is simply a linear mirroring transformation that mirrors Z_1 into Z_2 and vice versa. In 3-D, the bisecting surface is non-linear, and we do not define it explicitly; it is defined implicitly by the transformation we use for computing virtual vertices (Section 4.2).

As in the planar case, doubly virtual vertices might cause their virtual polygon to obscure the reflector. The solution in the planar case, a front clipping plane, can be generalized to non-planar reflectors by utilizing a second z-buffer containing the reflector's geometry. Every pixel generated during rendering of the virtual polygons will be tested twice: once against the ordinary z-buffer, in order to produce correct depth relationships between all virtual polygons, and once against the reflector z-buffer, to ensure that pixels in front of the reflectors are discarded.

A second z-buffer is not easy to define efficiently on today's graphics architectures. Alternatives that are currently more practical are: (1) do not do anything, anticipating that the obscuring pixels will fall outside the screen mask of the reflector, (2) approximate the reflector using six clipping planes, an option available on standard architectures, and (3) tessellate the scene so that mixed polygons are very small. Surprisingly, the first approach works well in the vast majority of cases, due to the way objects are usually positioned relative to each other and the way they are viewed. The second option reduces the problem but does not guarantee the resulting quality. The third option also reduces the problem, but requires more computations since there are more virtual vertices to compute. Tessellations are discussed in Sections 7 and 8.

4 The Explosion Map Acceleration Method

In some cases it is very efficient to compute the reflection subdivision and search it to find the cell in which a point lies (Section 5). In the general 3-D case, a faster indexing scheme is preferable.

In this section we describe an approximation method, the *explosion map*, which is a data structure for accelerating the computation of virtual vertices. It is prepared for each reflector separately, and recomputed whenever the viewpoint or the reflector are moved. The map is an image whose pixel values hold IDs of reflector triangles, and which represents a spherical 2-D cross section of the subdivision. To compute a virtual image of a scene point, we compute explosion map coordinates for it, thus yielding the ID of a specific triangle. The virtual image is computed using that triangle.

The explosion map is somewhat similar to a circular environment map [Haeberli93] in that it is an image in which a circle corresponds to the reflection directions (Figure 5(b)). However, it is unlike an environment map in that the latter contains renderings of other scene objects, while the explosion map contains only the reflector (Figure 5(a)). We next detail the computation (4.1) and utilization (4.2) of the map.

4.1 Computing an Explosion Map

An explosion map is a function of the tessellated reflector, the viewpoint, a 3-D sphere, and a desired resolution. The sphere should be centered at a point that is an intuitive 'center' of the reflector (as in environment mapping), and its radius should be large enough so that it bounds the reflector (actually, a sphere is not essential; we need any convex geometric object that approximates the reflector's shape). The map resolution should be large enough so that there are substantially more map pixels than reflector triangles. In practice, a resolution of 200^2 is sufficient when the reflector has been tessellated into several hundred triangles. The depth resolution of the map should have enough bits to hold unique IDs for all reflector vertices plus one more bit (needed to distinguish between ordinary triangles and extension polygons, defined below).

The basic operation in computing the map, **MapCoords**, involves

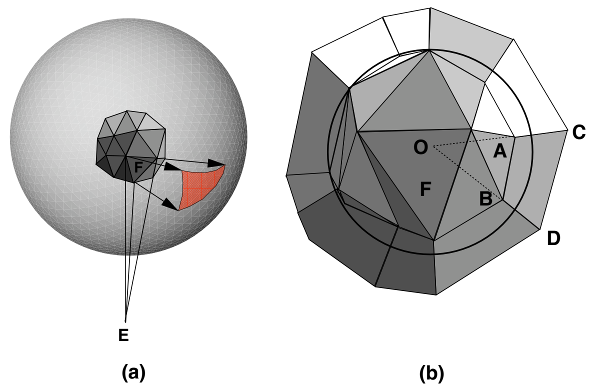


Figure 5 Explosion map: (a) reflection rays and intersection points on a bounding sphere; (b) the resulting map. *C* and *D* are extension vertices of *A* and *B*.

deriving the map coordinates $T = (t_x, t_y)$ corresponding to a normalized direction vector $N = (x, y, z)$ going from the center of the sphere to an arbitrary direction. If the resolution of the map is r^2, N is mapped to $T = (\frac{sx}{(2(z+1))^{1/2}} + s/2, \frac{sy}{(2(z+1))^{1/2}} + s/2)$, where s is a number a little smaller than r. This mapping is similar to that used for generating a circular environment map from a map rendered on the faces of a box [Haeberli93]. The pixels to which directions are mapped all fall inside a circle of radius $s/2$. The circle represents all possible reflection directions.

The map itself is computed as follows (Figure 6). For every front-facing reflector vertex, we compute map coordinates by intersecting its reflection ray with the sphere and calling **MapCoords** with the direction from the sphere's center to the intersection point (step 3). Back-facing vertices are denoted as such (step 4) to facilitate fast identification of profile triangles in step 6. For each front-facing reflector triangle (recall that a triangle is called front-facing if all its vertices are front-facing, and is called profile if only some of its vertices are front-facing), the corresponding triangle defined by the map coordinates is filled on the map, using its unique ID as color (step 5). Polygon fill can be done by the graphics hardware. For profile triangles, the normals of their back-facing vertices are projected in the direction of the viewer such that they are orthogonal to the line from the viewer through the vertex (step 7). The triangles thus become front-facing, and are now filled on the map as done for triangles that were front-facing originally (step 8).

So far, the interior of a map circle of radius $s/2$ has been partially filled, but not completely. This is due to the existence of the unreflected region and the fact that the filled map triangles are linear. As we explained in Section 3, we want the directions into the unreflected region to be filled on the map as well so that we could use it to compute doubly virtual vertices. To ensure that all directions are filled on the map, in **ExtendMap** the map is extended to cover the circle as follows. For each profile triangle V_{ijk} having two back-facing vertices (say V_i, V_j), we define an *extension polygon* E_{ij} in map coordinates and fill it with the ID of V_{ijk}. The vertices of E_{ij} are T_i, T_j, and extensions of each of these vertices in the direction away from the circle's center (in Figure 5(b), the extensions of vertices A, B are C, D). The extensions should be long enough so that the circle is completely covered. In practice, it is enough that the length of the segment from the center to each extended vertex is $0.6s$. Extension polygons effectively comprise an implicit representation of the bisecting surfaces Z explained in Section 3. Other methods for representing the unreflected region on the map are discussed in [Ofek98].

```
ExplosionMap (Reflector $R$, View $E$, Center $C$, Distance $d$,
        Resolution $r$):
(1) Let $S$ be a sphere centered at $C$ having radius $d$.
(2) Let $M$ be an image of size $r \times r$.
(3) For each reflector vertex $V_i$
        If $V_i$ is front-facing
            Let $R_i$ be the reflection ray of $V_i$.
            Let $I_i$ be the intersection of $R_i$ with $S$.
            Let $J_i$ be the normalized direction from $C$ to $I_i$.
            $T_i \leftarrow$ MapCoords ($J_i$, $r$).
        Else
(4)         Denote $V_i$ as back-facing.
(5) For each reflector triangle $V_{ijk}$
        If $V_{ijk}$ is front-facing
            Fill the triangle $T_i, T_j, T_k$ on $M$,
            using the ID of $V_{ijk}$ as the color.
(6)     Else if $V_{ijk}$ is a profile triangle
(7)         Fix its back-facing normals.
(8)         Compute and fill $T_i, T_j, T_k$ as before.
(9) ExtendMap ($R$, $M$).
```

Figure 6 Computing an explosion map.

4.2 Computing Virtual Vertices

The explosion map circle represents a mapping of all possible reflection directions. We use it to directly generate the final virtual vertex Q' corresponding to a potentially reflected scene vertex Q. For each reflector we compute two explosion maps: a *near map* and a *far map*. The near map is computed using a sphere that bounds the object but does not intersect any other object, and the far map is computed using a sphere that bounds all the scene. It is important to understand that although the topologies of the two maps are quite similar (because cells do not intersect each other), their geometries are different; reflection rays, which determine the geometry of map vertices, evolve non-linearly.

In addition to the explosion maps, we store a *hidden map* and an auxiliary z-buffer of the reflector. The hidden map is simply an item buffer of the visible mesh triangles. In other words, it is a discrete map in which a visible mesh triangle is mapped to a 2-D triangle filled by the ID of the mesh triangle. The map resolution can be smaller than that of the frame buffer (say, 200^2).

The basic operation needed is **MapToVirtualVertex**, whose arguments are a map M, a 3-D point Q and a corresponding map point I. Assume that the ID in $M(I)$ is that of an ordinary mesh triangle V (not an extension polygon) having 2-D vertices A, B, C (these are the T_i's computed in step 3 of Figure 6). The output is the virtual point Q'. The operation is implemented in three steps: (1) compute barycentric coordinates s, t of I relative to V by solving the two linear equations in two variables $(1-(s+t))A + sB + tC = I$; (2) use s, t as weights in a weighted average of the 3-D vertices and normals of V that yields a plane of reflection U; and (3) mirror Q across U to produce Q'. Note that negative barycentric coordinates are perfectly acceptable. The computation can be performed in integers or floating point, to reduce aliasing artifacts resulting from the discrete nature of the map. Extension polygons are handled similarly, using four bilinear coordinates instead of three. This treatment of extension polygons effectively implements the non-linear mirroring transformation of the unreflected region motivated in Section 3.

Computation of virtual vertices for a scene vertex Q is shown in Figure 7. We first determine if Q is hidden (steps 1, 2), by testing it in screen coordinates against the reflector's z-buffer. If it is, Q's virtual image is computed by the hidden map (step 3). Note that an obvious optimization here is to do this only for hidden vertices that belong to mixed polygons, since we don't need virtual images for polygons that are hidden completely.

```
**VirtualVertex** (Reflector $R$, Point $Q$, View $E$):
(1) Let $I$ be the screen coordinates of $Q$ (using $E$).
(2) If $Q$ is hidden by a mesh triangle $V$
(3)     Return **MapToVirtualVertex** ($HiddenMap, Q, I$).
(4) Let $c$ be the direction from the center of $R$ to $Q$.
(5) $T \leftarrow$ **MapCoords** ($c, r$).
(6) $Q'_n \leftarrow$ **MapToVirtualVertex** ($NearMap, Q, T$).
    $Q'_f \leftarrow$ **MapToVirtualVertex** ($FarMap, Q, T$).
(7) Let $d_n, d_f$ be the relative distances of $Q$ from
    the near and far spheres.
    Return $\frac{Q'_n/d_n + Q'_f/d_f}{1/d_n + 1/d_f}$.
```

Figure 7 Computing virtual vertices using the explosion and hidden maps.

When Q is not hidden we use the explosion maps. The normalized direction from the center of the reflector to Q is used to obtain map coordinates, in the same way used for creating the maps (steps 4, 5). Note that the map coordinates T are the same for both maps, but the triangle IDs found at T are different. In general, none of these triangles corresponds to the correct reflection cell in which Q is located, because we approximated the correct ray of reflection of Q by a ray from the center of the reflector (when higher accuracy is desired, we can use an improved approximation or locally search the correct cell [Ofek98].) Each of these triangles defines an auxiliary virtual vertex (step 6), and a weighted average of those is taken to obtain the final virtual vertex (step 7). There may be other ways to choose the weights than the obvious one shown. Figure 13 shows near and far explosion maps, in which polygon IDs are encoded by colors for visualization purposes.

5 Improved Efficiency for Linear Extrusions

For some common reflectors, it is possible to compute virtual vertices more efficiently than the explosion map, by directly utilizing the reflection subdivision to find the cell in which a scene point lies. Among these reflector are linear extrusions of planar curves (e.g., cylinders) and cones. For spheres, there is an efficient method that does not use the reflection subdivision at all. In general, if an implicit equation defining the reflector is available, the reflection point can be computed as in [Hanrahan92] (although this method is slow). Below we detail the case of an extruded reflector. The direct computation for cones and spheres is simple and given in [Ofek98].

Consider a 2-D reflection subdivision, as shown for example in Figure 4. We can optimize the step of identifying the cell in which a point lies by organizing the reflection cells in a hierarchy. Define $C_{i,j}$ to be the region bounded by reflection rays R_i, R_j and the line segment (V_i, V_j). Note that $C_{i,j}$ contains every cell $C_{k,r}, i \leq k < r \leq j$. Classifying a point with respect to a cell $C_{i,j}$ amounts to a few 'line side' tests, implemented by plugging the point into the line's equation and testing the sign of the result. If we find that the point is not contained in $C_{i,j}$, we know that it is outside all contained cells $C_{k,r}$. A binary search can thus be performed on the hierarchy. Note that there are no actual computations involved in generating the hierarchy, since it is implicitly represented by the numbering of the reflector vertices. A similar hierarchy can be defined for the hidden cells as well. Membership in the two (at most) unreflected cells can be tested easily. Consequently, the cell in which a point is located can be found using a small number ($O(\log n)$ where n is the reflector tessellation resolution) of 'line side' tests.

Suppose that the reflector is a linear extrusion of a convex 2-D planar curve. We can reduce the computation of a virtual vertex to 2-D by (1) projecting the viewer and all scene points onto the plane, (2) performing the 2-D computation, obtaining a line of reflection L_Q for each scene vertex, (3) extruding L_Q to 3-D to form a plane of reflection T, and (4) computing a final virtual vertex by mirroring the original vertex across T. The screen in Figure 17 is a linear extrusion of a convex planar curve.

6 Non-Convex Reflectors

Concave reflectors. The computations we perform for concave reflectors are identical to those for convex ones, but it is interesting to note that concave reflectors produce significantly more complicated visual results. In Figure 8 we see a viewer E in front of a concave reflector and three reflection rays. The reflection of an object located in region A (left) looks like an enlarged, deformed version of the object. The reflection of an object located in region B (middle) looks like an enlarged, deformed, upside-down version of the object. The reflection of objects located in region C (right) is utterly chaotic. This chaotic nature is inherent in the physics of reflections and is not an artifact of computations or approximations.

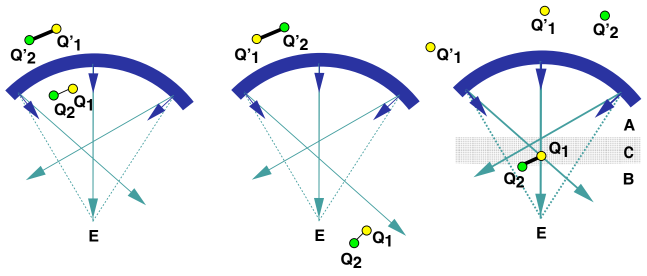


Figure 8 Behavior of reflections on concave reflectors.

Reflections of objects lying in regions A and B can be computed exactly as for convex reflectors, because in these regions the reflection subdivision is well-defined (since the reflection cells are disjoint). Reflections of objects lying in region C or intersecting that region are unpredictable and chaotic anyway, so almost any policy for computing virtual vertices will be satisfactory. In particular, we can simply use the value computed by the explosion map, thereby treating concave reflectors exactly as convex ones.

Figure 11 shows a concave reflector. On the right, we see the reflector, the reflection rays, a reflected planar object, and the computed virtual object, all these from a point of view different from the viewer's. On the left we see the final image from the viewer's point of view. The two explosion maps and the hidden map are shown at the bottom right. Note that reflected objects must be very close to this reflector to cross from region B to regions A or C.

Reflectors of mixed convexity. Reflectors that are neither convex nor concave should be decomposed into convex and concave parts. For many objects this can be done fully automatically [Spanguolo92]. Some polygonal surfaces contain saddles, resulting in a decomposition that is too fine. In such cases it is advised that users decompose the object manually. Note that the actual requirement is not of pure convexity of concavity, but rather that the reflection cells would not self-intersect in areas where reflected objects lie. Devising automatic algorithms that take this into consideration when decomposing the object is an interesting topic for future work. When manual decomposition is used, reflectors cannot dynamically change their shape in an arbitrary way, but the scene can still be dynamic. Figure 9 shows a reflector with a convex part (red) and a concave part (green). Note the seamless transition of the reflection image between the convex and concave parts.

7 Results

We have implemented our algorithms using OpenGL on SGIs running Irix and on PCs running Windows '95 and NT. Figure 12 shows a cylinder reflector modeled as a linear extrusion of a circle. Figure 13 demonstrates the effect of varying reflector tessellation resolution. The bottom part shows the near and far explosion maps. We see that using 128 triangles the reflection image already has an approximately correct geometric form, and that using 2048 rather than 512 triangles barely makes a difference. Figure 14 shows the effect of varying the tessellation of the reflected object (using 512 reflector triangles). A tessellation of 7x7 is sufficient. A lower resolution would suffice for objects farther away from the reflector,

Figure 16 shows a scene with four reflecting spheres, a table, and a window, rendered by our method (top) and by Rayshade, a well-known raytracer (bottom). A checkerboard texture was used in order to emphasize the reflections. The geometric shapes of the reflections in the two images are visually very similar. The texture in the bottom image is sharper because we use the graphics hardware for texture mapping.

On an SGI O_2, the top image required **less than a second**, and the bottom one required **50 seconds**. For Rayshade, we turned off shadows rays, highlights, and anti-aliasing, and we used a single sample per pixel and a manually tuned uniform grid as an acceleration scheme. Image resolution is 512^2.

Figure 15 shows the same scene, from a slightly different viewpoint and using real textures. The shadows are pre-computed textures. Figure 17 shows a reflecting TV modeled as an extrusion of a convex planar curve. Figure 18 shows recursive reflections on a planar mirror. Figure 19 shows a mask composed of several convex and concave pieces. Note the correct reflections of the red and green spheres on both 'cheeks' of the mask and on the nose. Figure 20 shows several reflecting polyhedra and a reflecting sphere. All of these scenes (except Figure 19) are displayed in real-time on an SGI Infinite Reality. We haven't tried the scene of Figure 19 on such a machine; on an SGI O_2, Figure 19 requires about a second with our method, and 1.5 minutes using Rayshade.

8 Discussion

The virtual objects method is the first method capable of accurately approximating reflections on curved objects at interactive rates. In this paper we presented the basic method for a single level of reflection and its implementation for general polygonal meshes and for linear extrusions. Clearly, the method possesses both advantages and disadvantages. We discuss these below, both in isolation and in the context of other methods.

Quality. In general, the quality produced by the method is satisfactory, especially for interactive use. The explosion map gives good results even for planar or nearly planar reflectors. Like any approximation method, ours might produce visible artifacts. The most noticeable ones occur when objects are not tessellated finely enough, in which case their reflections look too much like their real-world images and are not deformed according to the geometry of the reflector. In addition, reflections might be slightly translated inaccurately because we do not compute the exact explosion map cells to which vertices are mapped.

Other visible artifacts can be seen near the boundaries of the reflector, when the transformation used to create doubly virtual vertices is not a good approximation to the correct reflection. In this case the seam between convex and concave regions might be visible. Even in this case, reflections are self-consistent and do not exhibit holes.

When the reflector shape on the explosion map is far from convex, our heuristic for representing the unreflected region (extension polygons) might yield visible artifacts. Obviously, doubly virtual vertices might still hide the reflector when not using a second z-buffer. However, as we noted earlier, this usually does not happen because their screen images tend to fall outside the screen image of the reflector.

An attractive property of the method that has not been mentioned so far is that it supports interactive rendering of *refractions,* by using refraction rays instead of reflection rays. There are some additional differences, detailed in [Ofek98].

Tessellation strategies. As shown in Section 7, some tessellation of reflected objects is usually essential for providing sufficient accuracy. The finer the tessellation, the more accurate the reflections. At the same time, increasing the tessellation has an adverse impact on performance. These considerations are identical to those employed in interactive rendering of curved objects in general. There are two standard approaches: (1) usage of uniform tessellations, pre-computed such that quality is satisfactory, and (2) usage of hierarchical tessellations (levels of detail, etc).

Both approaches can be taken in our case as well. When the distances from a reflector to reflected objects and viewer remain approximately constant, we can pre-compute a uniform tessellation. The tessellation resolution of the reflected object should be chosen such that its virtual polygons cover several dozen pixels. Otherwise, hierarchical tessellations can be used. These can exhibit the same artifacts as when they are used for ordinary objects, e.g. discontinuities during animation. Note that the reflector tessellation resolution can be lower than that used when rendering its view-independent image. Using hierarchical tessellations is a topic for future work.

Performance. In the worst-case, all scene points can indeed be reflected on every convex part and every concave part of every reflector. Denote by r the number of visible reflectors and by $n(n')$ the number of vertices in the original (tessellated) scene. The time complexity of the method is $O(r \times n')$, which is thus worst-case optimal for a given degree of tessellation. For a single reflector, the step of computing the explosion and hidden maps is linear in the size of the reflector and is roughly equivalent to rendering the reflector three times at low resolution. The step of computing a virtual vertex for a scene vertex requires a relatively small *constant* number of operations. Moreover, the operations performed are highly regular, and are probably not too difficult to parallelize or implement in hardware. The cost of rendering virtual polygons is similar to rendering the whole scene. If deforming reflectors are desired, they should be subdivided into convex and concave parts on each frame, which costs time linear in their size. Naturally, as the depth complexity of the reflection image increases, the time complexity of our method diverges from the optimal.

The scenes shown in this paper run interactively (1-30 frames per second) on an SGI O_2 workstation. This performance was achieved *without any optimization*; in particular, no method for culling objects that cannot be reflected has been used. On today's systems, without further optimizations the number of reflected objects cannot be much larger than shown while still guaranteeing interactive performance.

Comparison to other methods. We can compare our method to environment mapping or ray tracing, which are currently the only techniques capable of computing reflections on curved objects. Both visual accuracy and efficiency should be considered.

Environment mapping is relatively accurate only when reflected objects are relatively far from the reflector and when the curvature of the reflector is not large. When the scene is static, time complexity is linear in the size of the reflector, because the map can be precomputed. This is in general much faster than our method. When the scene is dynamic, the map must be recomputed on each frame for each reflector. This also holds when only the viewer changes, unless the special hardware of [Voorhies94] is used. Complexity is $r \times n$, which is closer to our method but still more efficient. To what degree depends on the amount of tessellation. However, environment mapping simply does not provide realistic accuracy. Seeing reflections of objects that are nearby as if they are very far creates an uneasy feeling and definitely cannot be qualified as realistic.

Ray tracing obviously produces higher quality images than our method and supports a wider range of illumination phenomena. Regarding efficiency, the relevant characteristics of our method are: (1) it operates at the object level rather than the pixel level (we have an object and we want to know where it is reflected, rather than having the point of reflection and seeking an object), (2) it transforms the problem into one that standard graphics systems can handle, (3) it transforms the computation into a local one involving a single reflector-reflected pair, instead of the global ray tracing computation ('find the nearest object'); global visibility relationships are automatically handled by the z-buffer, and (4) it uses both the CPU and the graphics system, dividing (but not necessarily balancing) the load between them. When these properties are significant, our method is more efficient than ray tracing. Ray tracing can be expected to perform better when (1) reflected objects do not cover many pixels, (2) there are many curved reflectors, or (3) the depth complexity of the reflected images is large. It may or may not be faster when there is no graphics hardware. Note that our method scales much better than ray tracing to larger image resolutions, while ray tracing scales better with scene depth complexity.

It is very difficult to predict the point from which ray tracing is more efficient. On the relatively simple scenes shown in this paper, the method is at least an order of magnitude more efficient than Rayshade, a well-known available ray tracer, even when it uses a manually tuned acceleration scheme.

Future work. Both efficiency and quality issues should be further investigated. Efficiency issues include: acceleration using global scene organization techniques, hierarchical tessellations, possible hardware implementation, acceleration using time coherence, and usage of the method to accelerate other illumination methods. Quality issues include refining the initial approximation given by the explosion map, improved methods for filling the unreflected region on the map, using the method for rendering refractions, automatic decomposition of reflectors of mixed convexity, quantifying the degree of error introduced by our approximations, and additional levels of recursive reflections.

Conclusion. We feel that correct reflections from small objects are not very important. Such reflections, reflections on complex mixed convexity objects, and reflections of distant objects can be convincingly emulated using environment mapping. High quality reflections are therefore needed for relatively large objects with relatively uniform convexity (or concavity). A typical scene does not contain too many curved objects like these. As a result, although the time complexity of the method is theoretically quadratic in the number of reflectors, in practice its complexity is linear in the size of the scene (it can be sub-linear if scene databases are used for culling objects). Applicability will increase with increases in processing power and graphics hardware. Even today, there are many applications in which the number of objects in the scene is less important than the rendering quality. In these cases, our method is at least an order of magnitude faster than ray tracing and provides higher visual quality than environment mapping.

Our experience is that interacting with scenes containing reflections is immensely more enjoyable than with scenes without reflections. Reflections bring dull and lifeless scenes to life.

Acknowledgements. We thank Dani Lischinski for commenting on a draft of this paper and for fruitful discussions. We also thank Amichai Nitsan for his involvement in part of the implementation. Lastly, we warmly thank Leo Krieger for his continuous support.

References

[Blinn76] Blinn, J., Newell, M., Texture and reflection in computer generated images. *Comm. ACM,* 19:542–546, 1976.

[Diefenbach97] Diefenbach, P.J., Badler, N.I., Multi-pass pipeline rendering: realism for dynamic environments. Proceedings, *1997 Symposium on Interactive 3D Graphics*, ACM Press, 1997.

[Foley90] Foley, J.D., Van Dam A., Feiner, S.K., Hughes, J.F., Computer Graphics: Principles and Practice, 2nd ed., Addison-Wesley, 1990.

[Glassner89] Glassner, A. (ed), An Introduction to Ray Tracing. Academic Press, 1989.

[Greene86] Greene, N., Environment mapping and other applications of world projections. *IEEE CG&A*, 6(11), Nov. 1986.

[Haeberli93] Haeberli, P., Segal, M., Texture mapping as a fundamental drawing primitive. Proceedings, *Fourth Eurographics Workshop on Rendering,* Cohen, Puech, Sillion (eds), 1993, pp. 259–266.

[Hall96] Hall, T., Tutorial on planar mirrors in OpenGL, posted to comp.graphics.api.opengl, Aug. 1996.

[Hanrahan92] Hanrahan, P., Mitchell, D., Illumination from curved reflectors. Proceedings, Siggraph '92, ACM Press, pp. 283–291.

[Heckbert84] Heckbert, P.S., Hanrahan, P., Beam tracing polygonal objects. *Computer Graphics,* 18:119–127, 1984 (Siggraph '84).

[Jansen93] Jansen, F.W., Realism in real-time? Proceedings, *Fourth Eurographics Workshop on Rendering,* Cohen, Puech, Sillion (eds), 1993.

[McReynolds96] McReynolds, T., Blythe, D., Programming with OpenGL: Advanced Rendering, course #23, Siggraph '96.

[Ofek98] Ofek, E., Modeling and Rendering 3-D Objects. Ph.D. thesis, Institute of Computer Science, The Hebrew University, 1998.

[Rushmeier86] Rushmeier, H.E., Extending the radiosity method to transmitting and specularly reflecting surfaces. Masters's thesis, Cornell University, 1986.

[Segal92] Segal, M., Korobkin, C., van Widenfelt, R., Foran, J., Haeberli, P., Fast shadows and lighting effects using texture mapping. *Computer Graphics*, 26:249–252, 1992 (Siggraph '92).

[Sillion89] Sillion, F., Puech, C., A general two-pass method integrating specular and diffuse reflection. *Computer Graphics*, 23(3):335–344 (Siggraph '89).

[Spanguolo92] Spanguolo, M., Polyhedral surface decomposition based on curvature analysis. In: *Modern Geometric Computing for Visualization,* T.L. Kunii and Y. Shinagawa (Eds.), Springer-Verlag, 1992.

[Voorhies94] Voorhies, D., Foran, J., Reflection vector shading hardware. Proceedings, Siggraph '94, ACM Press, pp. 163–166.

[Wallace87] Wallace, J.R., Cohen, M.F., Greenberg, D.P, A two-pass solution to the rendering equation: a synthesis of ray tracing and radiosity methods. *Computer Graphics,* 21:311–320, 1987 (Siggraph '87).

[Whitted80] Whitted, T., An improved illumination model for shaded display. *Comm. of the ACM,* 23(6):343–349, 1980.

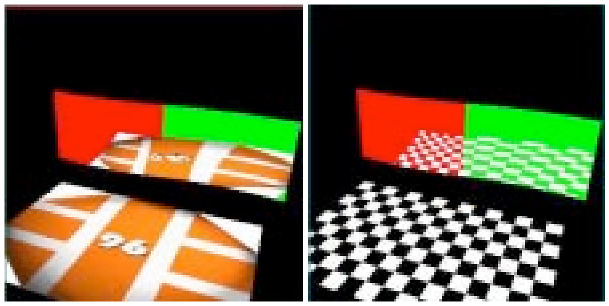


Fig. 9: Mixed convexity reflector, with seamless reflections.

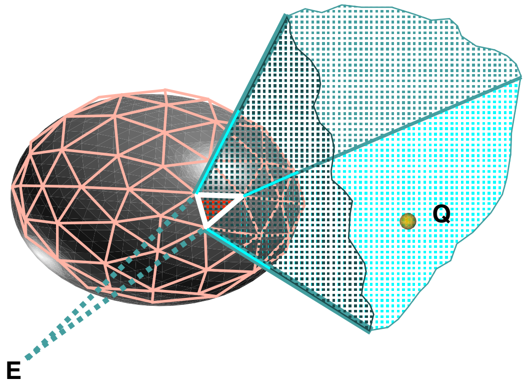


Fig. 10: A 3 D reflected cell.

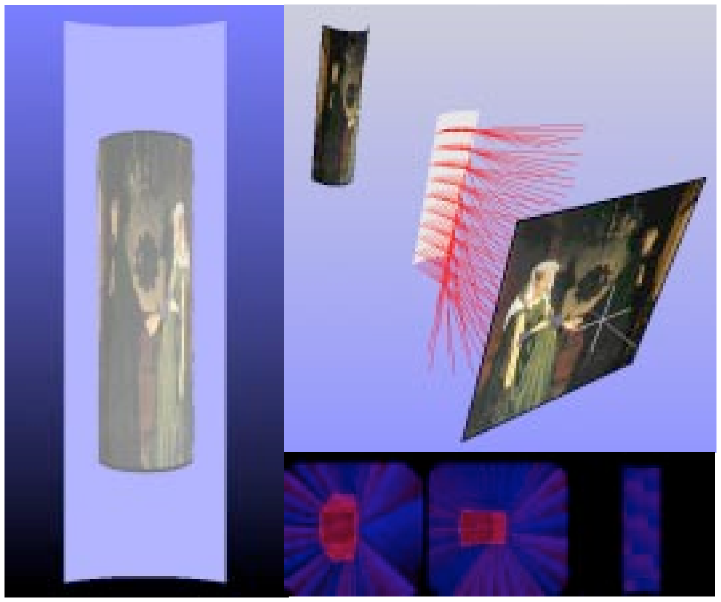


Fig. 11: Virtual object and reflection rays.

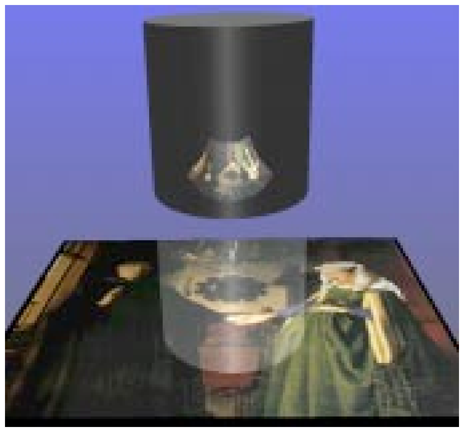


Fig. 12: Linear extrusion.

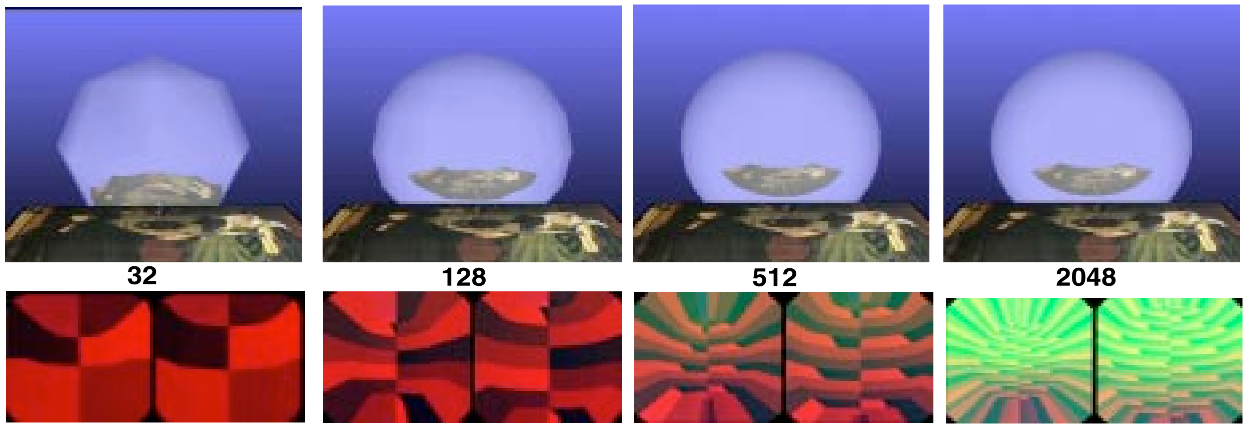


Fig. 13: Varying the tessellation resolution of the reflector.

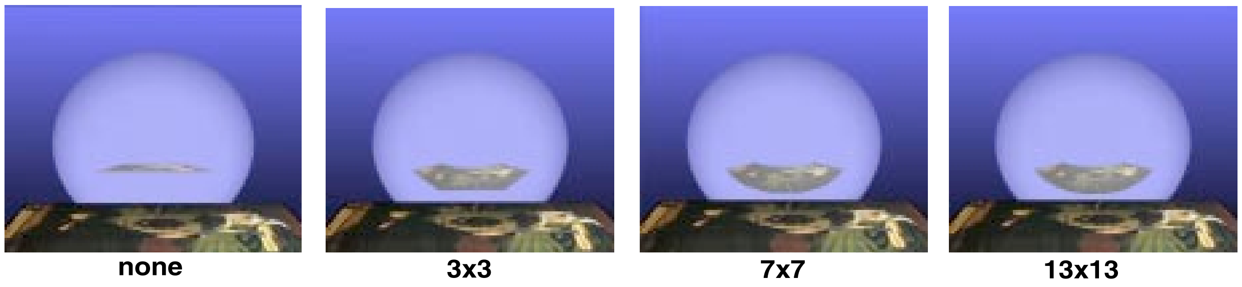


Fig. 14: Varying the tessellation resolution of the reflected object.

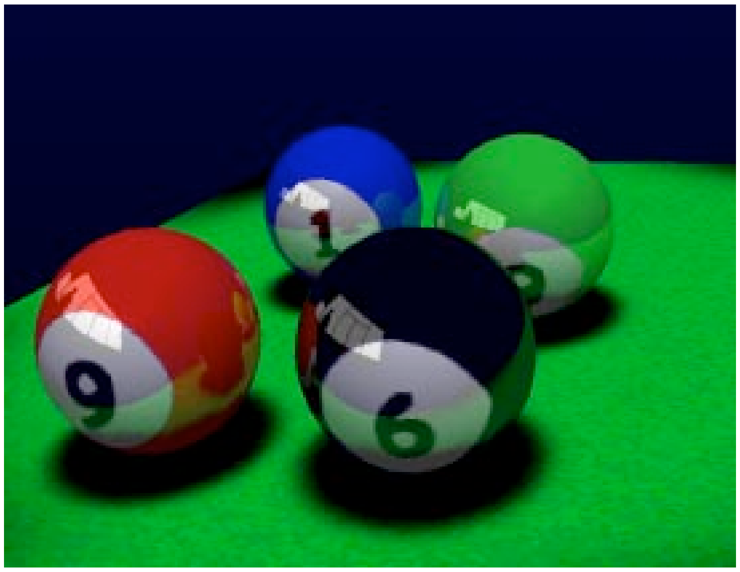


Fig. 15: Four reflecting spheres.

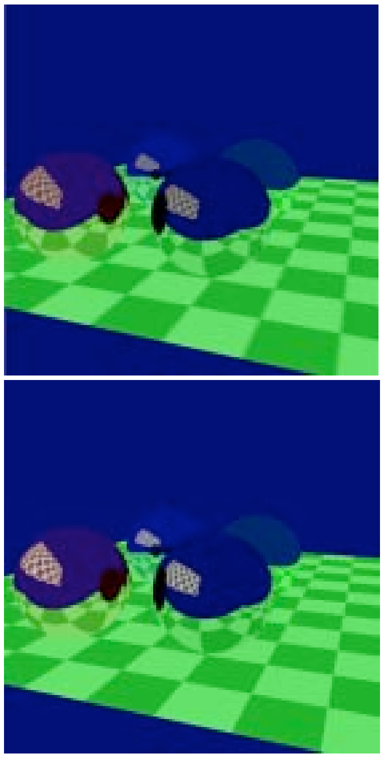


Fig. 16: Top: our method. Bottom: Rayshade.

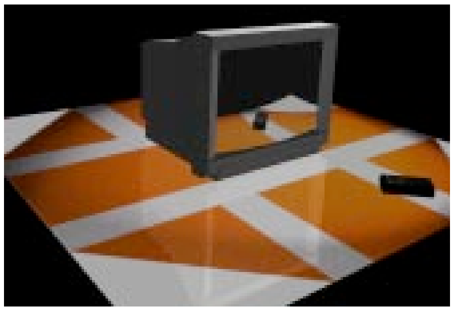


Fig. 17: TV.

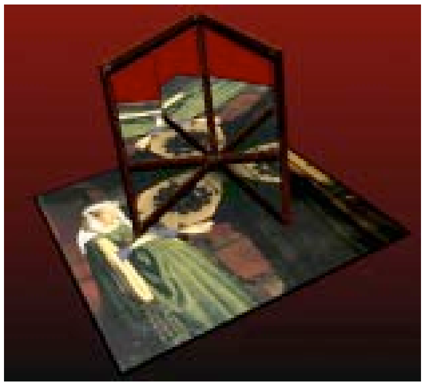


Fig. 18: Recursive reflections.

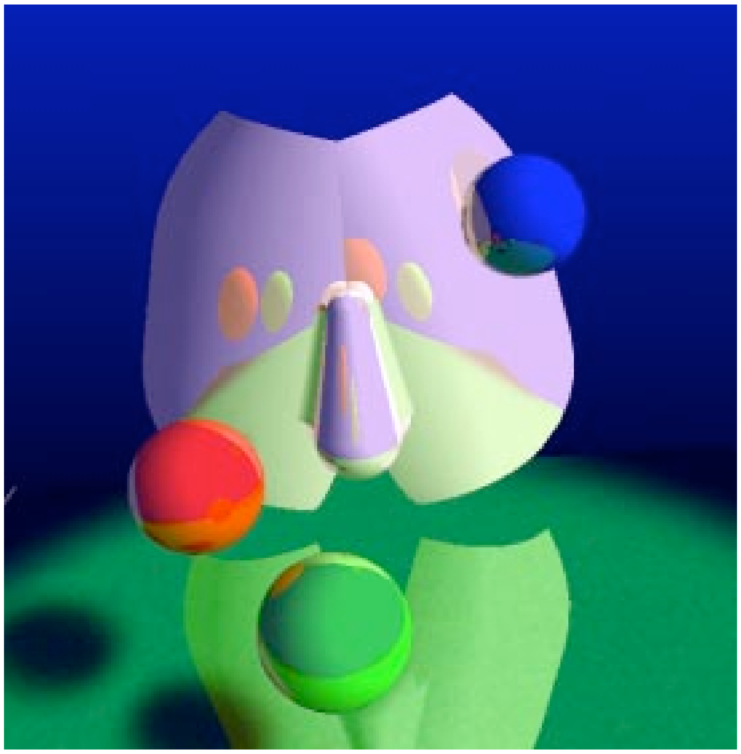


Fig. 19: Reflector containing several convex and concave pieces.

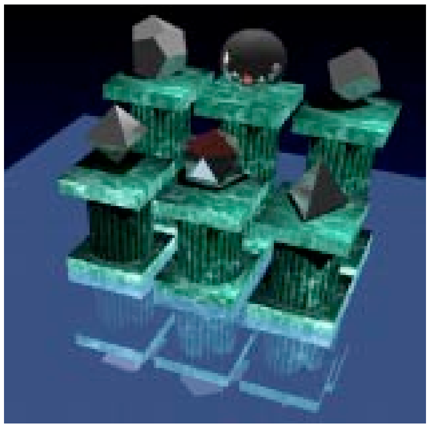


Fig. 20: Polyhedra and sphere.

View-independent Environment Maps

Wolfgang Heidrich and Hans-Peter Seidel

Computer Graphics Group
University of Erlangen
{heidrich,seidel}@informatik.uni-erlangen.de

Abstract

Environment maps are widely used for approximating reflections in hardware-accelerated rendering applications. Unfortunately, the parameterizations for environment maps used in today's graphics hardware severely undersample certain directions, and can thus not be used from multiple viewing directions. Other parameterizations exist, but require operations that would be too expensive for hardware implementations.

In this paper we introduce an inexpensive new parameterization for environment maps that allows us to reuse the environment map for any given viewing direction. We describe how, under certain restrictions, these maps can be used today in standard OpenGL implementations. Furthermore, we explore how OpenGL could be extended to support this kind of environment map more directly.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors; I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and framebuffer operations; I.3.6 [Computer Graphics]: Methodology and Techniques—Standards I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, Shading, Shadowing and Texture I.4.1 [Image Processing and Computer Vision]: Digitization and Image Capture—Sampling

Keywords: Environment mapping, OpenGL

1 Introduction

The basic idea of environment maps is striking [1]: if a reflecting object is small compared to its distance from the environment, the incoming illumination on the surface really only depends on the direction of the reflected ray. Its origin, that is the actual position on the surface, can be neglected. Therefore, the incoming illumination at the object can be precomputed and stored in a 2-dimensional texture map.

If the parameterization for this texture map is cleverly chosen, the illumination for reflections off the surface can be looked up very efficiently. Of course, the assumption of a small object compared to the environment often does not hold, but environment maps are a good compromise between rendering quality, and the need to store the full, 4-dimensional radiance field on the surface.

Both offline [3, 9] and interactive, hardware-based renderers [8] have used this implementation of reflections, often with amazing results.

Given the above description of environment maps, one would think that it should be possible to use a single map for all viewing positions and directions. After all, the environment map is supposed to contain information about illumination from *all* directions. Thus, it should be possible to modify the lookup process in order to extract the correct information for all possible points of view.

In reality, however, this is not quite true. The parameterization used in most of today's graphics hardware exhibits a singularity as well as areas of extremely poor sampling. As a consequence, this form of environment map cannot be used for any viewing direction except the one for which it was originally generated.

This leaves us with a situation, where the environment map has to be re-generated for each frame even in simple applications such as walkthroughs. Other parameterizations exist (see Section 2), but require operations that would be too expensive for hardware implementations. In this paper we introduce a parameterization for environment maps that uses only simple operations (additions, multiplications and matrix operations), but provides a good enough sampling so that *one* map can be used for *all* viewing directions.

In the following, we first discuss existing parameterizations for environment maps and their deficiencies. Then, in Section 3, we introduce our new parameterization, and describe how it can be used on contemporary graphics hardware under certain restrictions (Section 5). Finally, we propose a simple extension to the texture coordinate generation of OpenGL that would add direct support for our method (Section 6), and we present results of our implementation in (Section 7).

2 Previous Work

The parameterization used most commonly in computer graphics hardware today, are *spherical environment maps* [8]. It is based on the simple analogy of a small, perfectly mirroring ball centered around the object. The image that an orthographic camera sees when looking at this ball from a certain viewing direction is the environment map. An example environment map from the center of a colored cube is shown in Figure 1.

The sampling rate of this map is maximal for directions opposing the viewing direction (that is, objects behind the viewer), and goes towards zero for directions close to the viewing direction. Moreover, there is a singularity the in viewing direction, since all points where the viewing vector is tangential to the sphere show the same point of the

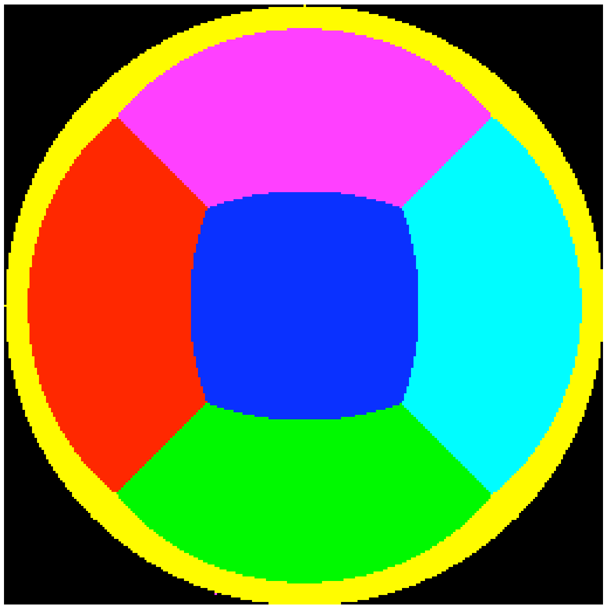


Figure 1: A spherical environment map from the center of a colored cube. Note the bad sampling of the cube face directly in front of the observer (light gray).

environment.

With these properties, it is clear that this parameterization is not suitable for viewing directions other than the original one. The major reason why it is used anyway, is that the lookup can be computed efficiently with simple operations in hardware. The parameterization proposed in this paper solves the sampling problems while at the same time maintaining the simplicity of the lookup process.

Another parameterization are *latitude-longitude maps* [9]. Here, the s, and t parameters of the texture map are interpreted as the latitude and longitude, respectively, with respect to a certain viewing direction. Apart from the fact that these maps are severely oversampled around the poles, the lookup process involves the computation of inverse trigonometric functions, which is inappropriate for hardware implementations.

Finally, *cubical environment maps* [2, 10] consist of 6 independent perspective images from the center of a cube through each of its faces. The sampling of these maps is fairly good, as the sampling rates for the directions differ by a factor of $3\sqrt{3} \approx 5.2$. Also, the lookup process within each of the 6 images is inexpensive. However, the difficulty is to decide which of the six images to use for the lookup. This requires several conditional jumps, and interpolation of texture coordinates is difficult for polygons containing vertices in more than one image. Because of these problems cubical environment maps are difficult and expensive to implement in hardware, although they are quite widespread in software renderers (e.g. [9]).

Many interactive systems initially obtain the illumination as a cubical environment map, and then resample this information into a spherical environment map. There are two ways this can be done. The first is to re-render the cubical map for every frame, so that the cube is always aligned with the current viewing direction. Of course this is slow if the environment contains complex geometry. The other method is to generate the cubical map only once, and then re-compute the mapping from the cubical to the spherical map for each frame. This, however, makes the resampling step more expensive, and can lead to numerical problems around the singularity.

In both cases, the resampling can be performed as a multipass algorithm in hardware, using morphing and texture mapping. Yet, the bandwidth imposed by this method onto the graphics system is quite large: the six textures from the cubical representation have to be loaded into texture memory, and the resulting image has to be transferred from the framebuffer into texture RAM or even into main memory.

3 A New Parameterization

The parameterization we use is based on an analogy similar to the one used to describe spherical environment maps. Assume that the reflecting object lies in the origin, and that the viewing direction is along the negative z axis. The image seen by an orthographic camera when looking at the paraboloid

$$f(x,y) = \frac{1}{2} - \frac{1}{2}(x^2+y^2), \quad x^2+y^2 \le 1 \tag{1}$$

contains the information about the hemisphere facing towards the viewer. The complete environment is stored in two separate textures, each containing the information of one hemisphere. The geometry is depicted in Figure 2.

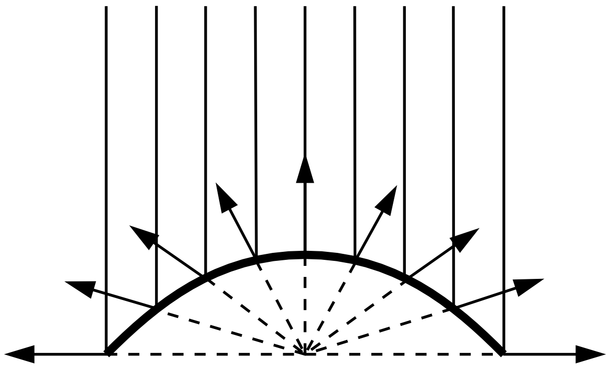


Figure 2: The rays of an orthographic camera reflected on a paraboloid sample a complete hemisphere of directions.

It should be noted that this parameterization has recently been introduced by Nayar [6, 5] in a different context. Nayar actually built a lens and camera system that is capable of capturing this sort of image from the real world. Besides raytracing and resampling of cubical environment maps, this is actually one way of acquiring maps in the proposed format. Since two of these cameras can be attached back to back [5], it is possible to create full 360° images of real world scenes.

The geometry described above has some interesting properties. Firstly, the reflection rays in each point of the paraboloid all intersect in a single point, the origin (see dashed lines in Figure 2). This means that the resulting image can indeed be used as an environment map for an object in the origin.

Secondly, the sampling rate varies by a factor of 4 over the complete image, as depicted in Figure 3. Pixels in the outer regions of the map cover only 1/4 of the solid angle covered by center pixels. This means that directions perpendicular to the viewing direction are sampled at a higher rate than directions parallel to the viewing direction. Depending on how we select mipmap levels, the factor of 4 in the sampling rate corresponds to one or two levels difference, which is quite acceptable. In particular this is better than the sampling of cubical environment maps.

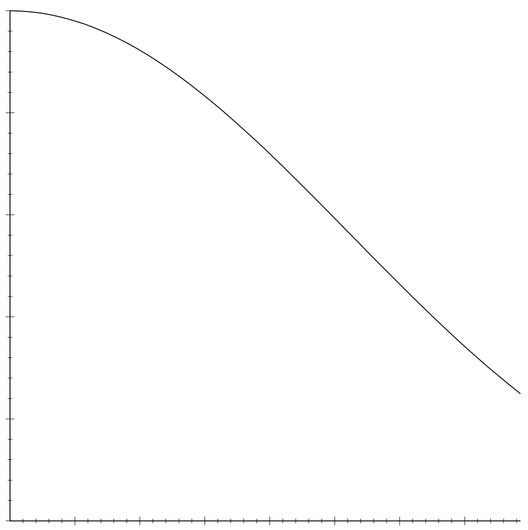


Figure 3: The change of solid angle ω covered by a single pixel versus the angle θ between the viewing direction and the reflected ray. The sampling rate varies by a factor of 4 over the environment map. Directions perpendicular to the viewing direction are sampled at a higher rate than directions parallel to the viewing direction.

Figure 4 shows the two images comprising an environment map for the simple scene used in Figure 1. The top image represents the hemisphere facing towards the camera, while the bottom image represents the hemisphere facing away from it.

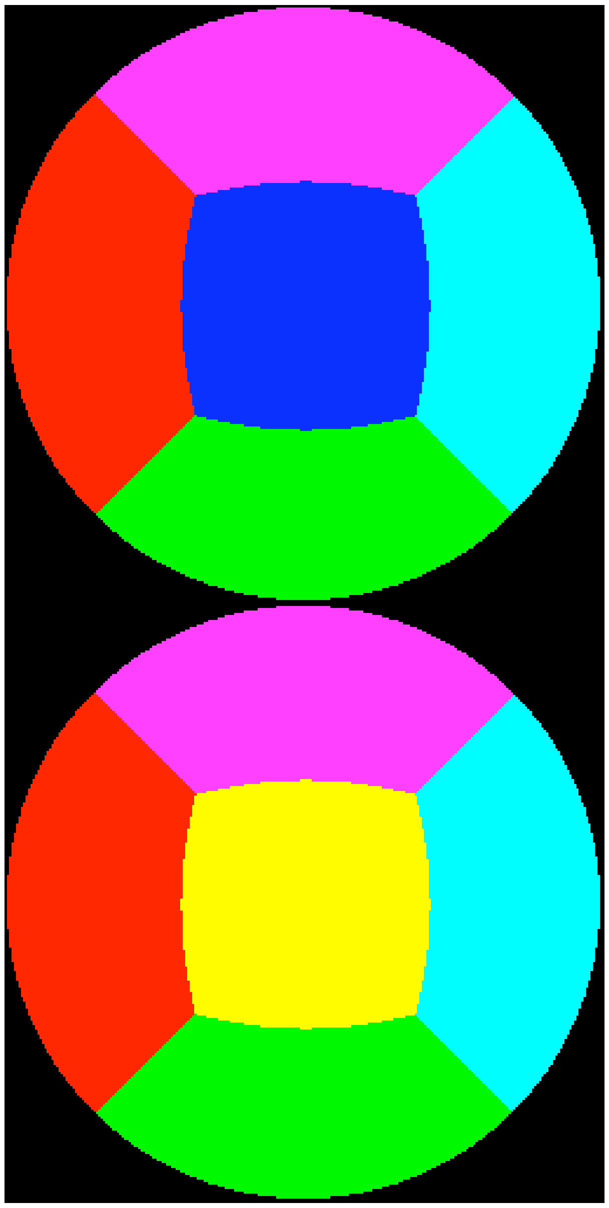


Figure 4: The two textures comprising an environment map for an object in the center of a colored cube.

4 Lookups from Arbitrary Viewing Positions

In the following, we describe the math behind the lookup of a reflection value from an arbitrary viewing position. We assume that environment maps are specified relative to a coordinate system, where the reflecting object lies in the origin, and the map is generated for a viewing direction of $\mathbf{d}_o = (0, 0, -1)^T$. It is not necessary that this coordinate system represents the object space of the reflecting object, although this would be an obvious choice. However, it is important that the transformation between this space and eye space is a rigid body transformation, as this means that vectors do not have to be normalized after transformation. To simplify the notation, we will in the following use the term “object space” for this space.

In the following, $\mathbf{v}_e$ denotes the normalized vector from the eye point to the point on the surface, while the vector $\mathbf{n}_e = (n_{e,x}, n_{e,y}, n_{e,z})^T$ is the normal of the surface point in eye space. Furthermore, the (affine) model-view matrix is given as $\mathbf{M}$. This means, that the normal vector in eye space $\mathbf{n}_e$ is really the transformation $\mathbf{M}^{-T} \cdot \mathbf{n}_o$ of some normal vector in object space. If $\mathbf{M}$ is a rigid body transformation, and $\mathbf{n}_o$ was normalized, then so is $\mathbf{n}_e$. The reflection vector in eye space is then given as

$$\mathbf{r}_e = \mathbf{v}_e + 2 \langle \mathbf{n}_e, -\mathbf{v}_e \rangle \cdot \mathbf{n}_e. \quad (2)$$

Transforming this vector with the inverse of $\mathbf{M}$ yields the reflection vector in object space:

$$\mathbf{r}_o = \mathbf{M}^{-1} \cdot \mathbf{r}_e. \quad (3)$$

The illumination for this vector in object space is stored somewhere in one of the two images. More specifically, if the z component of this vector is positive, the vector is facing towards the viewer, and thus the value is in the first image, otherwise it is in the second. Let us, for the moment, consider the first case.

$\mathbf{r}_o$ is the reflection of the constant vector $\mathbf{d}_o = (0, 0, -1)^T$ in some point (x, y, z) on the paraboloid:

$$\mathbf{r}_o = \mathbf{d}_o + 2\langle \mathbf{n}, -\mathbf{d}_o \rangle \cdot \mathbf{n}, \tag{4}$$

where $\mathbf{n}$ is the normal at that point of the paraboloid. Due to the formula of the paraboloid from Equation 1, this normal vector happens to be

$$\mathbf{n} = \frac{1}{\sqrt{x^2 + y^2 + 1}} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \tag{5}$$

Combining Equations 4 and 5 yields

$$\mathbf{d}_o - \mathbf{r}_o = 2\langle \mathbf{n}, \mathbf{v} \rangle \mathbf{n} = \begin{pmatrix} k \cdot x \\ k \cdot y \\ k \end{pmatrix}. \tag{6}$$

In summary, this means that x and y, which can be directly mapped to texture coordinates, can be computed by calculating the reflection vector in eye space (Equation 2), transforming it back into object space (Equation 3), subtracting it from the (constant) vector $\mathbf{d}_o$ (Equation 6), and finally dividing by the z component of the resulting vector.

The second case, where the z component of the reflection vector in object space is negative, can be handled similarly, except that $-\mathbf{d}$ has to be used in Equation 6, and that the resulting values are $-x$ and $-y$.

5 OpenGL Implementation

An interesting observation of the above equations is that almost all the required operations are linear. There are two exceptions. The first is the calculation of the reflection vector in eye space (Equation 2), which is quadratic in the components of the normal vector $\mathbf{n}_e$. The second exception is the division at the end, which can, however, be implemented as a perspective divide.

Given the reflection vector $\mathbf{r}_e$ in eye coordinates, the transformations for the frontfacing part of the environment can be written in homogeneous coordinates as follows:

$$\begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix} = \mathbf{P} \cdot \mathbf{S} \cdot (\mathbf{M}_l)^{-1} \cdot \begin{bmatrix} r_{e,x} \\ r_{e,y} \\ r_{e,z} \\ 1 \end{bmatrix}, \tag{7}$$

where

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

is a projective transformation that divides by the z component,

$$\mathbf{S} = \begin{bmatrix} -1 & 0 & 0 & d_{o,x} \\ 0 & -1 & 0 & d_{o,y} \\ 0 & 0 & 1 & d_{o,z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

computes $\mathbf{d}_o - \mathbf{r}_o$, and $\mathbf{M}_l$ is the linear part of the affine transformation $\mathbf{M}$. Another matrix is required for mapping x and y into the interval $[0, 1]$ for the use as texture coordinates:

$$\begin{bmatrix} s \\ t \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix}$$

Similar transformations can be derived for the backfacing parts of the environment. These matrices can be used as texture matrices, if $\mathbf{r}_e$ is specified as the texture coordinate for the vertex. Note that $\mathbf{r}_e$ changes from vertex to vertex, while the matrices remain constant.

Due to the non-linearity of the reflection vector, $\mathbf{r}_e$ has to be computed in software. This is the step that corresponds to the automatic generation of texture coordinates for spherical environment maps in OpenGL (`glTexGen`). Actually, this process can be further simplified by assuming that the vector $\mathbf{v}$ from the eye to the object point is constant. This is true, if the object is far away from the camera, compared to its size, or if the camera is orthographic. Otherwise, the assumption breaks down, which is particularly noticeable on flat objects.

What remains to be done is to combine frontfacing and backfacing regions of the environment into a single image. To this end we use OpenGL's alpha test feature. In the two texture maps, we mark those pixels inside the circle $x^2 + y^2 \leq 1$ with an alpha value of 1, the pixels outside the circle with an alpha value of 0. Then the algorithm works as follows:

```
glAlphaFunc( GL_EQUAL, 1.0 );
glEnable( GL_ALPHA_TEST );
glMatrixMode( GL_TEXTURE );

glBindTexture( GL_TEXTURE_2D,
               frontFacingMap );
glLoadMatrix( frontFacingMatrix );
draw object with r_o as texture coord.

glBindTexture( GL_TEXTURE_2D,
               backFacingMap );
glLoadMatrix( backFacingMatrix );
draw object with r_o as texture coord.
```

The important point here is that backfacing vectors $\mathbf{r}_o$ will result in texture coordinates $x^2 + y^2 > 1$ while the matrix for the frontfacing part is active, and will thus not be rendered. Similarly frontfacing vectors will not be rendered while the matrix for the backfacing part is active.

6 Extending OpenGL

While the method described in Section 5 works, and is also quite fast (see Section 7), it is not the best one could hope for. Firstly, since the texture coordinates have to be generated in software, it is not possible to use display lists[1]. Secondly, many vectors required to compute the reflected vector

[1]Actually, since the texture coordinates are identical for both passes of our method, display lists can be used to render the two passes within one frame, but they cannot be reused for other frames.

$\mathbf{r}_e$ are already available further down the pipeline (that is, when OpenGL texture coordinate generation takes place), but are not easily available in software.

For example, the normal vector in a vertex is typically only known in object space. In order to compute $\mathbf{n}_e$, this vector has to be transformed by hand, although the transformed normal is later being transformed by the hardware anyway (for lighting calculations).

Interestingly, the texture coordinates generated for spherical environment maps are the x and y components of the halfway vector between the reflection vector $\mathbf{r}_e$ and the negative viewing direction $(0,0,1)^T$ [8]. Thus, current OpenGL implementations essentially already require the computation of $\mathbf{r}_e$ for environment mapping.

We would like to emphasize that this computation of the reflection vector is really necessary, even with the standard OpenGL spherical environment maps. On first sight, one could think that for this parameterization, it would be possible to directly use the x and y components of the surface normal as texture coordinates. This, however, would only yield the desired result for orthographic views. For perspective views it would lead to sever artifacts, such as flat mirroring surfaces being colored in a single, uniform color.

Because OpenGL implementations already compute the reflection vector, the changes necessary to fully support our parameterization are minimal. All that is required, is a new mode for texture coordinate generation that directly stores $\mathbf{r}_e$ into the s, t, and r texture coordinates, and sets q to 1.

Furthermore, and independent of this proposal, it would be possible get rid the second rendering pass by allowing for multiple textures to be bound at the same time. OpenGL extensions for this purpose are currently being discussed by the OpenGL architecture review board.

7 Results

We have implemented the software-based method described in Section 5, and tested it with several scenes. Figure 5 shows two images making up one environment map. These images where generated using ray-casting. The circles indicate regions with $x^2+y^2 \leq 1$. The regions outside the circles are do not really belong to the map, but have been generated during the ray-casting step by extending the paraboloid to the domain $-1 \leq x, y \leq 1$. We found it useful to have an additional ring of one or two pixels available outside the actual circle, in order to avoid seams in regions where front- and backfacing regions touch.

Figure 6 shows a sphere to which this environment map has been applied. The images have been taken from different viewpoints, but with the same environment map. This image can be rendered in full screen (1280×1024) at about 15 frames per second on an SGI O2 and at > 20 frames per second on an SGI RealityEngine2. The tessellation used for this sphere was 72×72 quadrilaterals.

A closeup of the seams between frontfacing and backfacing regions of the environment map can be seen in Figure 7. In the left image, the curve indicates this seam, which hard to detect in the right image.

Finally, Figure 8 shows a torus with the environment map applied. Here we used a tessellation of 144×72, and the timings where 13 frames per second on the O2, and > 20 frames per second on the RealityEngine2, again at full screen resolution.

Based on the discussion in Section 6, we are confident, that these times could be further improved, if some minor extensions were made to the texture coordinate generation mechanism of OpenGL.

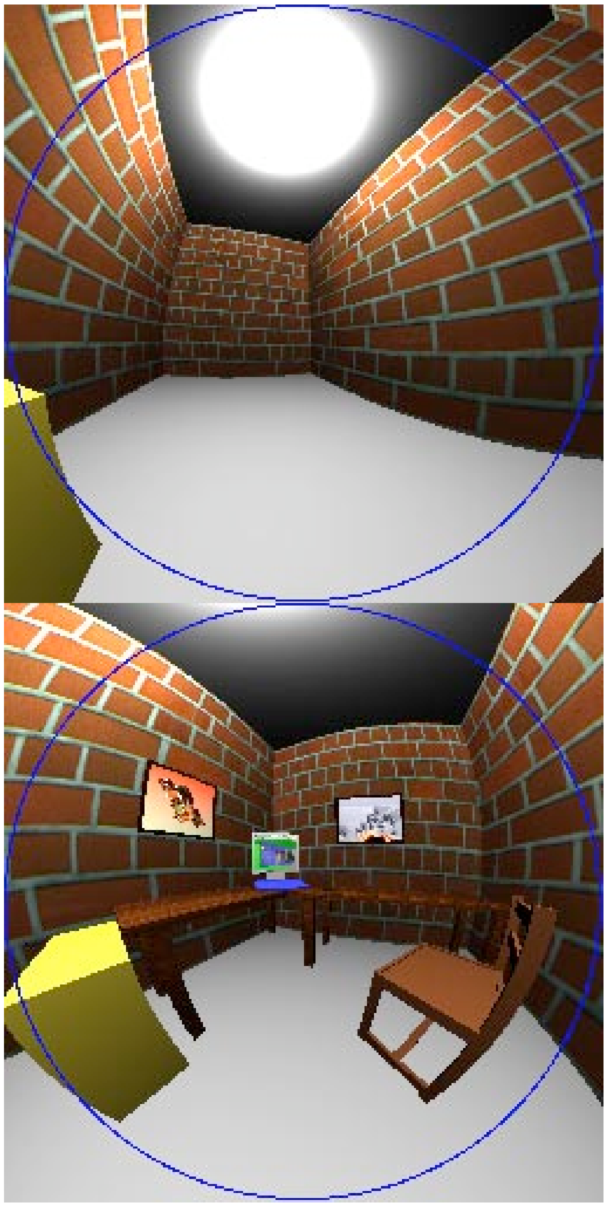


Figure 5: The two textures of the environment map for an object in the center of an office scene.

8 Conclusions

In this paper, we have introduced a novel parameterization for environment maps, which allows us to reuse them from multiple viewpoints. This allows us to generate walkthroughs of scenes with reflecting objects without the need to recompute environment maps for each frame.

We have shown how the new parameterization can be used today in standard OpenGL implementations. Although this method is partly based on a software algorithm, we have

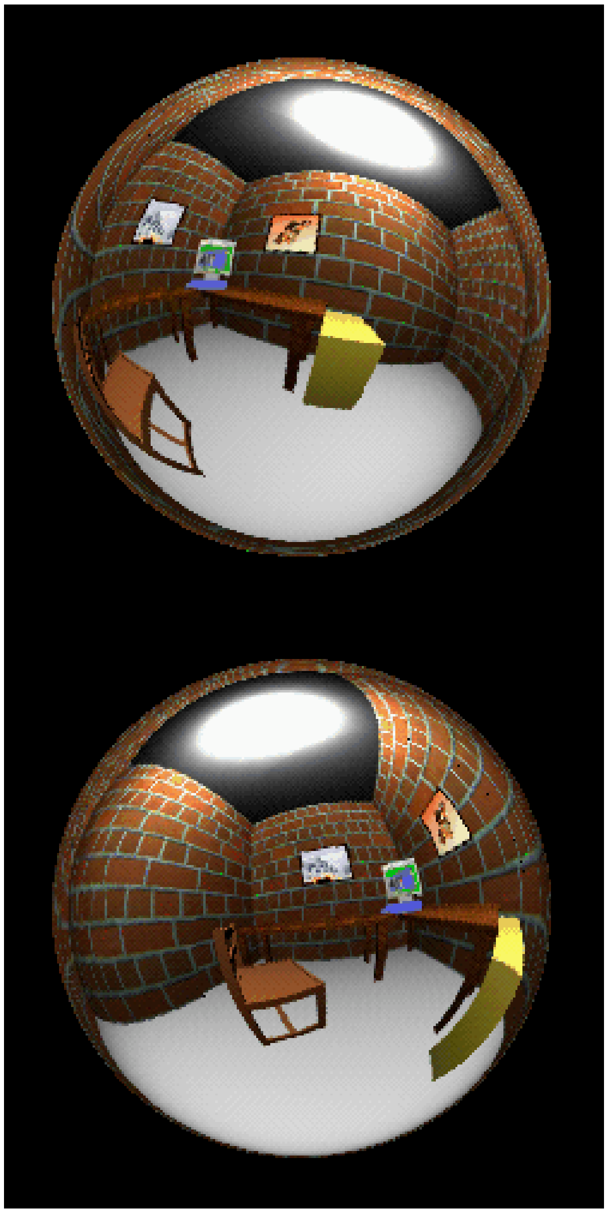


Figure 6: The environment map from Figure 5 applied to a sphere seen from two different viewpoints.

demonstrated it to be efficient enough for many practical purposes.

Further performance improvements are possible by adding some direct support for our parameterization into the OpenGL API. Only minimal changes would be necessary for this support, and they would be backwards compatible to the current interface.

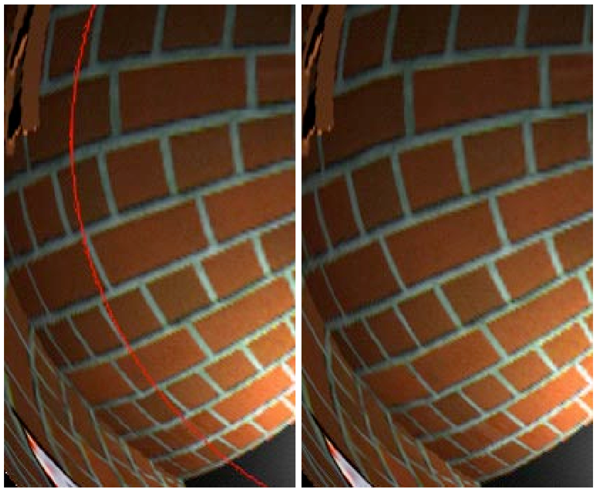


Figure 7: A closeup of the seams between the frontfacing and the backfacing regions of the environment map. The curve in the left image indicates the location of the seam, which is hard to detect in the right image.

References

[1] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19:542–546, 1976.

[2] Ned Greene. Applications of world projections. In M. Green, editor, *Proceedings of Graphics Interface '86*, pages 108–114, May 1986.

[3] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculation. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):289–298, August 1990.

[4] Shree Nayar. Omnicamera home page. Available from http://www.cs.columbia.edu/CAVE/omnicam, 1997.

[5] Shree Nayar. Omnidirectional image sensing. Invited Talk at the 1998 Workshop on Image-Based Modeling and Rendering, March 1998.

[6] Shree K. Nayar. Catadioptric omnidirectional camera. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 482–488, June 1997.

[7] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison Wesley, 1993.

[8] OpenGL ARB. *OpenGL Specification, Version 1.1*, 1995.

[9] Pixar. *The RenderMan Interface, Version 3.1*. Pixar, San Rafael, CA, September 1989.

[10] Douglas Voorhies and Jim Foran. Reflection vector shading hardware. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 163–166, July 1994.

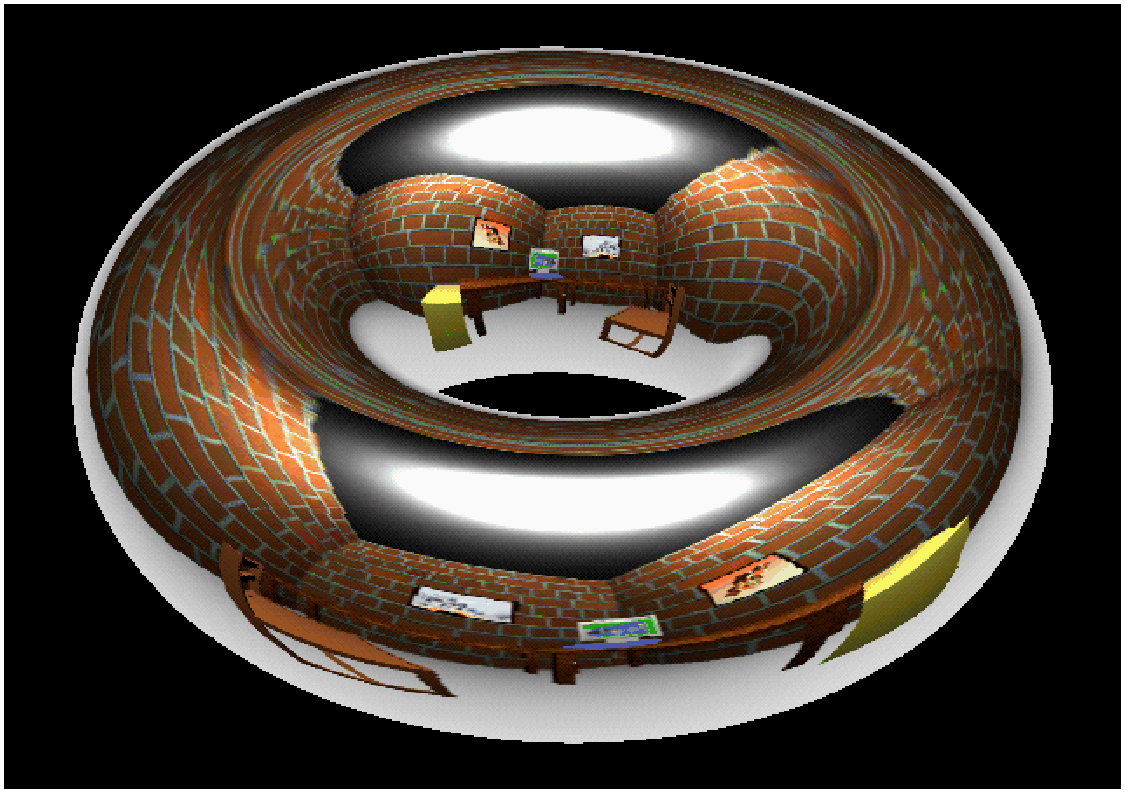


Figure 8: A torus with view-independent environment mapping, rendered using OpenGL

.

Fast Shadows and Lighting Effects Using Texture Mapping

Mark Segal
Carl Korobkin
Rolf van Widenfelt
Jim Foran
Paul Haeberli
Silicon Graphics Computer Systems*

Abstract

Generating images of texture mapped geometry requires projecting surfaces onto a two-dimensional screen. If this projection involves perspective, then a division must be performed at each pixel of the projected surface in order to correctly calculate texture map coordinates.

We show how a simple extension to perspective-correct texture mapping can be used to create various lighting effects. These include arbitrary projection of two-dimensional images onto geometry, realistic spotlights, and generation of shadows using shadow maps[10]. These effects are obtained in real time using hardware that performs correct texture mapping.

CR Categories and Subject Descriptors: I.3.3 **[Computer Graphics]:** Picture/Image Generation; I.3.7 **[Computer Graphics]:** Three-Dimensional Graphics and Realism - *color, shading, shadowing, and texture*

Additional Key Words and Phrases: lighting, texture mapping

1 Introduction

Producing an image of a three-dimensional scene requires finding the projection of that scene onto a two-dimensional screen. In the case of a scene consisting of texture mapped surfaces, this involves not only determining where the projected points of the surfaces should appear on the screen, but also which portions of the texture image should be associated with the projected points.

*2011 N. Shoreline Blvd., Mountain View, CA 94043

If the image of the three-dimensional scene is to appear realistic, then the projection from three to two dimensions must be a perspective projection. Typically, a complex scene is converted to polygons before projection. The projected vertices of these polygons determine boundary edges of projected polygons.

Scan conversion uses iteration to enumerate pixels on the screen that are covered by each polygon. This iteration in the plane of projection introduces a homogeneous variation into the parameters that index the texture of a projected polygon. We call these parameters *texture coordinates*. If the homogeneous variation is ignored in favor of a simpler linear iteration, incorrect images are produced that can lead to objectionable effects such as texture "swimming" during scene animation[5]. Correct interpolation of texture coordinates requires each to be divided by a common denominator for each pixel of a projected texture mapped polygon[6].

We examine the general situation in which a texture is mapped onto a surface via a projection, after which the surface is projected onto a two dimensional viewing screen. This is like projecting a slide of some scene onto an arbitrarily oriented surface, which is then viewed from some viewpoint (see Figure 1). It turns out that handling this situation during texture coordinate iteration is essentially no different from the more usual case in which a texture is mapped linearly onto a polygon. We use *projective textures* to simulate spotlights and generate shadows using a method that is well-suited to graphics hardware that performs divisions to obtain correct texture coordinates.

2 Mathematical Preliminaries

To aid in describing the iteration process, we introduce four coordinate systems. The *clip* coordinate system is a homogeneous representation of three-dimensional space, with x, y, z, and w coordinates. The origin of this coordinate system is the viewpoint. We use the term clip coordinate system because it is this system in which clipping is often carried out. The *screen* co-

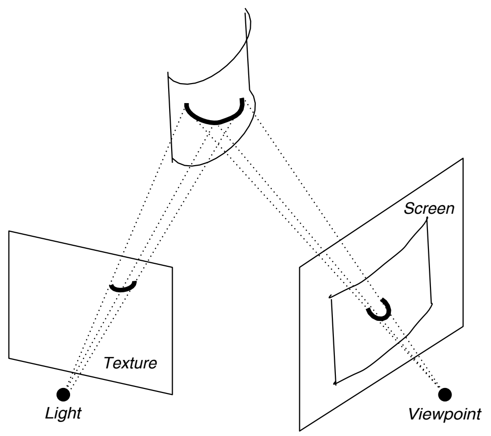


Figure 1. Viewing a projected texture.

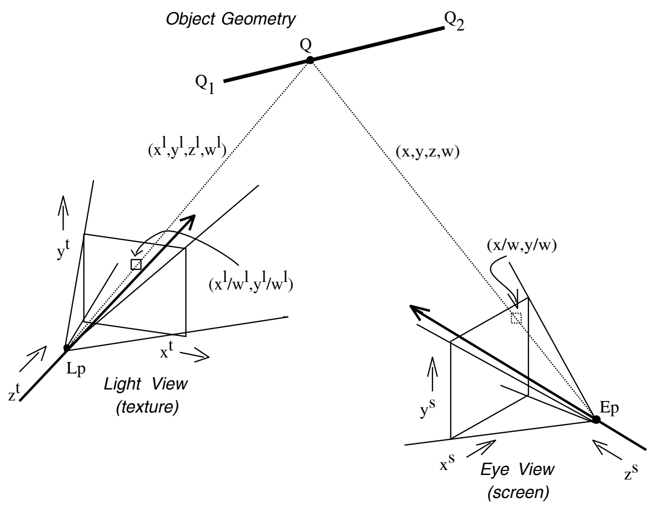


Figure 2. Object geometry in the light and clip coordinate systems.

ordinate system represents the two-dimensional screen with two coordinates. These are obtained from clip coordinates by dividing x and y by w, so that screen coordinates are given by $x^s = x/w$ and $y^s = y/w$ (the s superscript indicates screen coordinates). The *light* coordinate system is a second homogeneous coordinate system with coordinates x^l, y^l, z^l, and w^l; the origin of this system is at the light source. Finally, the *texture* coordinate system corresponds to a texture, which may represent a slide through which the light shines. Texture coordinates are given by $x^t = x^l/w^l$ and $y^t = y^l/w^l$ (we shall also find a use for $z^t = z^l/w^l$). Given (x^s, y^s), a point on a scan-converted polygon, our goal is to find its corresponding texture coordinates, (x^t, y^t).

Figure 2 shows a line segment in the clip coordinate system and its projection onto the two-dimensional screen. This line segment represents a span between two edges of a polygon. In clip coordinates, the endpoints of the line segment are given by

$$\mathbf{Q}_1 = (x_1, y_1, z_1, w_1) \qquad \text{and} \qquad \mathbf{Q}_2 = (x_2, y_2, z_2, w_2).$$

A point $\mathbf{Q}$ along the line segment can be written in clip coordinates as

$$\mathbf{Q} = (1-t)\mathbf{Q}_1 + t\mathbf{Q}_2 \tag{1}$$

for some $t \in [0,1]$. In screen coordinates, we write the corresponding projected point as

$$\mathbf{Q}^s = (1-t^s)\mathbf{Q}_1^s + t^s\mathbf{Q}_2^s \tag{2}$$

where $\mathbf{Q}_1^s = \mathbf{Q}_1/w_1$ and $\mathbf{Q}_2^s = \mathbf{Q}_2/w_2$.

To find the light coordinates of $\mathbf{Q}$ given $\mathbf{Q}^s$, we must find the value of t corresponding to t^s (in general $t \neq t^s$). This is accomplished by noting that

$$\mathbf{Q}^s = (1-t^s)\mathbf{Q}_1/w_1 + t^s\mathbf{Q}_2/w_2 = \frac{(1-t)\mathbf{Q}_1 + t\mathbf{Q}_2}{(1-t)w_1 + tw_2} \tag{3}$$

and solving for t. This is most easily achieved by choosing a and b such that $1-t^s = a/(a+b)$ and $t^s = b/(a+b)$; we also choose A and B such that $(1-t) = A/(A+B)$ and $t = B/(A+B)$. Equation 3 becomes

$$\mathbf{Q}^s = \frac{a\mathbf{Q}_1/w_1 + b\mathbf{Q}_2/w_2}{(a+b)} = \frac{A\mathbf{Q}_1 + B\mathbf{Q}_2}{Aw_1 + Bw_2}. \tag{4}$$

It is easily verified that $A = aw_2$ and $B = bw_1$ satisfy this equation, allowing us to obtain t and thus $\mathbf{Q}$.

Because the relationship between light coordinates and clip coordinates is affine (linear plus translation), there is a homogeneous matrix M that relates them:

$$\mathbf{Q}^l = M\mathbf{Q} = \frac{A}{A+B}\mathbf{Q}_1^l + \frac{B}{A+B}\mathbf{Q}_2^l \tag{5}$$

where $\mathbf{Q}_1^l = (x_1^l, y_1^l, z_1^l, w_1^l)$ and $\mathbf{Q}_2^l = (x_2^l, y_2^l, z_2^l, w_2^l)$ are the light coordinates of the points given by $\mathbf{Q}_1$ and $\mathbf{Q}_2$ in clip coordinates.

We finally obtain

$$\begin{aligned}\mathbf{Q}^t &= \mathbf{Q}^l/w^l \\ &= \frac{A\mathbf{Q}_1^l + B\mathbf{Q}_2^l}{Aw_1^l + Bw_2^l} \\ &= \frac{a\mathbf{Q}_1^l/w_1 + b\mathbf{Q}_2^l/w_2}{a(w_1^l/w_1) + b(w_2^l/w_2)}.\end{aligned} \tag{6}$$

Equation 6 gives the texture coordinates corresponding to a linearly interpolated point along a line segment in screen coordinates. To obtain these coordinates at a pixel, we must linearly interpolate x^l/w, y^l/w, and w^l/w, and divide at each pixel to obtain

$$x^l/w^l = \frac{x^l/w}{w^l/w} \qquad \text{and} \qquad y^l/w^l = \frac{y^l/w}{w^l/w}. \tag{7}$$

(For an alternate derivation of this result, see [6].)

If w^l is constant across a polygon, then Equation 7 becomes

$$s = \frac{s/w}{1/w} \qquad \text{and} \qquad t = \frac{t/w}{1/w}, \tag{8}$$

where we have set $s = x^l/w^l$ and $t = y^l/w^l$. Equation 8 governs the iteration of texture coordinates that have simply been assigned to polygon vertices. It still implies a division for each pixel contained in a polygon. The more general situation of a projected texture implied by Equation 7 requires only that the divisor be w^l/w instead of $1/w$.

3 Applications

To make the various coordinates in the following examples concrete, we introduce one more coordinate system: the *world* coordinate system. This is the coordinate system in which the three-dimensional model of the scene is described. There are thus two transformation matrices of interest: M_c transforms world coordinates to clip coordinates, and M_l transforms world coordinates to light coordinates. Iteration proceeds across projected polygon line segments according to equation 6 to obtain texture coordinates (x^t, y^t) for each pixel on the screen.

3.1 Slide Projector

One application of projective texture mapping consists of viewing the projection of a slide or movie on an arbitrary surface[9][2]. In this case, the texture represents the slide or movie. We describe a multi-pass drawing algorithm to simulate film projection.

Each pass entails scan-converting every polygon in the scene. Scan-conversion yields a series of screen points and corresponding texture points for each polygon. Associated with each screen point is a color and z-value, denoted c and z, respectively. Associated with each corresponding texture point is a color and z-value, denoted c_τ and z_τ. These values are used to modify corresponding values in a framebuffer of pixels. Each pixel, denoted p, also has an associated color and z-value, denoted c_p and z_p.

A color consists of several indepenedent components (e.g. red, green, and blue). Addition or multiplication of two colors indicates addition or multiplication of each corresponding pair of components (each component may be taken to lie in the range $[0, 1]$).

Assume that z_p is initialized to some large value for all p, and that c_p is initialized to some fixed ambient scene color for all p. The slide projection algorithm consists of three passes; for each scan-converted point in each pass, these actions are performed:

Pass 1 If $z < z_p$, then $z_p \leftarrow z$ *(hidden surface removal)*

Pass 2 If $z = z_p$, then $c_p \leftarrow c_p + c_\tau$ *(illumination)*

Pass 3 Set $c_p = c \cdot c_p$ *(final rendering)*

Pass 1 is a z-buffering step that sets z_p for each pixel. Pass 2 increases the brightness of each pixel according to the projected spotlight shape; the test ensures that portions of the scene visible from the eye point are brightened by the texture image only once (occlusions are not considered). The effects of multiple film projections may be incorporated by repeating Pass 2 several times, modifying M_l and the light coordinates appropriately on each pass. Pass 3 draws the scene, modulating the color of each pixel by the corresponding color of the projected texture image. Effects of standard (i.e. non-projective) texture mapping may be incorporated in this pass. Current Silicon Graphics hardware is capable of performing each pass at approximately 10^5 polygons per second.

Figure 3 shows a slide projected onto a scene. The left image shows the texture map; the right image shows the scene illuminated by both ambient light and the projected slide. The projected image may also be made to have a particular focal plane by rendering the scene several times and using an accumulation buffer as described in [4].

The same configuration can transform an image cast on one projection plane into a distinct projection plane. Consider, for instance, a photograph of a building's facade taken from some position. The effect of viewing the facade from arbitrary positions can be achieved by projecting the photograph back onto the building's facade and then viewing the scene from a different vantage point. This effect is useful in walk-throughs or fly-bys; texture mapping can be used to simulate buildings and distant scenery viewed from any viewpoint[1][7].

3.2 Spotlights

A similar technique can be used to simulate the effects of spotlight illumination on a scene. In this case the texture represents an intensity map of a cross-section of the spotlight's beam. That is, it is as if an opaque screen were placed in front of a spotlight and the intensity at each point on the screen recorded. Any conceivable spot shape may be accommodated. In addition, distortion effects, such as those attributed to a shield or a lens, may be incorporated into the texture map image.

Angular attenuation of illumination is incorporated into the intensity texture map of the spot source. Attenuation due to distance may be approximated by applying a function of the depth values $z^t = z^l/w^l$ iterated along with the texture coordinates (x^t, y^t) at each pixel in the image.

This method of illuminating a scene with a spotlight is useful for many real-time simulation applications, such

Figure 3. Simulating a slide projector.

as aircraft landing lights, directable aircraft taxi lights, and automotive headlights.

3.3 Fast, Accurate Shadows

Another application of this technique is to produce shadows cast from any number of point light sources. We follow the method described by Williams[10], but in a way that exploits available texture mapping hardware.

First, an image of the scene is rendered from the viewpoint of the light source. The purpose of this rendering is to obtain depth values in light coordinates for the scene with hidden surfaces removed. The depth values are the values of z^l/w^l at each pixel in the image. The array of z^t values corresponding to the hidden surface-removed image are then placed into a texture map, which will be used as a *shadow map*[10][8]. We refer to a value in this texture map as z_τ.

The generated texture map is used in a three-pass rendering process. This process uses an additional framebuffer value α_p in the range $[0, 1]$. The initial conditions are the same as those for the slide projector algorithm.

Pass 1 If $z < z_p$, then $z_p \leftarrow z$, $c_p \leftarrow c$ *(hidden surface removal)*

Pass 2 If $z_\tau = z^t$, then $\alpha_p \leftarrow 1$; else $\alpha_p \leftarrow 0$ *(shadow testing)*

Pass 3 $c_p \leftarrow c_p + (c \text{ modulated by } \alpha_p)$ *(final rendering)*

Pass 1 produces a hidden surface-removed image of the scene using only ambient illumination. If the two values in the comparison in Pass 2 are equal, then the point represented by p is visible from the light and so is not in shadow; otherwise, it is in shadow. Pass 3, drawn with full illumination, brightens portions of the scene that are not in shadow.

In practice, the comparison in Pass 2 is replaced with $z_\tau > z^t + \epsilon$, where ϵ is a bias. See [8] for factors governing the selection of ϵ.

This technique requires that the mechanism for setting α_p be based on the result of a comparison between a value stored in the texture map and the iterated z^t. For accuracy, it also requires that the texture map be capable of representing large z_τ. Our latest hardware posseses these capabilites, and can perform each of the above passes at the rate of at least 10^5 polygons per second.

Correct illumination from multiple colored lights may be produced by performing multiple passes. The shadow effect may also be combined with the spotlight effect described above, as shown in Figure 4. The left image in this figure is the shadow map. The center image is the spotlight intensity map. The right image shows the effects of incorporating both spotlight and shadow effects into a scene.

This technique differs from the hardware implementation described in [3]. It uses existing texture mapping hardware to create shadows, instead of drawing extruded shadow volumes for each polygon in the scene. In addition, percentage closer filtering [8] is easily supported.

4 Conclusions

Projecting a texture image onto a scene from some light source is no more expensive to compute than simple texture mapping in which texture coordinates are assinged to polygon vertices. Both require a single division per-pixel for each texture coordinate; accounting for the texture projection simply modifies the divisor.

Viewing a texture projected onto a three-dimensional scene is a useful technique for simulating a number of effects, including projecting images, spotlight illumination, and shadows. If hardware is available to perform texture mapping and the per-pixel division it requires, then these effects can be obtained with no performance penalty.

Figure 4. Generating shadows using a shadow map.

Acknowledgements

Many thanks to Derrick Burns for help with the texture coordinate iteration equations. Thanks also to Tom Davis for useful discussions. Dan Baum provided helpful suggestions for the spotlight implementation. Software Systems provided some of the textures used in Figure 3.

References

[1] Robert N. Devich and Frederick M. Weinhaus. Image perspective transformations. *SPIE*, 238, 1980.

[2] Julie O'B. Dorsey, Francois X. Sillion, and Donald P. Greenberg. Design and simulation of opera lighting and projection effects. In *Proceedings of SIGGRAPH '91*, pages 41–50, 1991.

[3] Henry Fuchs, Jack Goldfeather, and Jeff P. Hultquist, et al. Fast spheres, shadows, textures, transparencies, and image enhancements in pixels-planes. In *Proceedings of SIGGRAPH '85*, pages 111–120, 1985.

[4] Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Proceedings of SIGGRAPH '90*, pages 309–318, 1990.

[5] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, UC Berkeley, June 1989.

[6] Paul S. Heckbert and Henry P. Moreton. Interpolation for polygon texture mapping and shading. In David F. Rogers and Rae A. Earnshaw, editors, *State of the Art in Computer Graphics: Visualization and Modeling*, pages 101–111. Springer-Verlag, 1991.

[7] Kazufumi Kaneda, Eihachiro Nakamae, Tomoyuki Nishita, Hideo Tanaka, and Takao Noguchi. Three dimensional terrain modeling and display for environmental assessment. In *Proceedings of SIGGRAPH '89*, pages 207–214, 1989.

[8] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *Proceedings of SIGGRAPH '87*, pages 283–291, 1987.

[9] Steve Upstill. *The RenderMan Companion*, pages 371–374. Addison Wesley, 1990.

[10] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of SIGGRAPH '78*, pages 270–274, 1978.

Simulating Soft Shadows with Graphics Hardware

Paul S. Heckbert and Michael Herf

January 15, 1997
CMU-CS-97-104

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

email: ph@cs.cmu.edu, herf+@cmu.edu
World Wide Web: http://www.cs.cmu.edu/∼ph

This paper was written in April 1996. An abbreviated version appeared in [Michael Herf and Paul S. Heckbert, Fast Soft Shadows, *Visual Proceedings, SIGGRAPH 96*, Aug. 1996, p. 145].

Abstract

This paper describes an algorithm for simulating soft shadows at interactive rates using graphics hardware. On current graphics workstations, the technique can calculate the soft shadows cast by moving, complex objects onto multiple planar surfaces in about a second. In a static, diffuse scene, these high quality shadows can then be displayed at 30 Hz, independent of the number and size of the light sources.

For a diffuse scene, the method precomputes a *radiance texture* that captures the shadows and other brightness variations on each polygon. The texture for each polygon is computed by creating registered projections of the scene onto the polygon from multiple sample points on each light source, and averaging the resulting hard shadow images to compute a soft shadow image. After this precomputation, soft shadows in a static scene can be displayed in real-time with simple texture mapping of the radiance textures. All pixel operations employed by the algorithm are supported in hardware by existing graphics workstations. The technique can be generalized for the simulation of shadows on specular surfaces.

This work was supported by NSF Young Investigator award CCR-9357763. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF or the U.S. government.

Keywords: penumbra, texture mapping, graphics workstation, interaction, real-time, SGI Reality Engine.

1 Introduction

Shadows are both an important visual cue for the perception of spatial relationships and an essential component of realistic images. Shadows differ according to the type of light source causing them: point light sources yield hard shadows, while linear and area (also known as *extended*) light sources generally yield soft shadows with an umbra (fully shadowed region) and penumbra (partially shadowed region).

The real world contains mostly soft shadows due to the finite size of sky light, the sun, and light bulbs, yet most computer graphics rendering software simulates only hard shadows, if it simulates shadows at all. Excessive sharpness of shadow edges is often a telltale sign that a picture is computer generated.

Shadows are even less commonly simulated with hardware rendering. Current graphics workstations, such as Silicon Graphics (SGI) and Hewlett Packard (HP) machines, provide z-buffer hardware that supports real-time rendering of fairly complex scenes. Such machines are wonderful tools for computer aided design and visualization. Shadows are seldom simulated on such machines, however, because existing algorithms are not general enough, or they require too much time or memory. The shadow algorithms most suitable for interaction on graphics workstations have a cost per frame proportional to the number of point light sources. While such algorithms are practical for one or two light sources, they are impractical for a large number of sources or the approximation of extended sources.

We present here a new algorithm that computes the soft shadows due to extended light sources. The algorithm exploits graphics hardware for fast projective (perspective) transformation, clipping, scan conversion, texture mapping, visibility testing, and image averaging. The hardware is used both to compute the shading on the surfaces and to display it, using texture mapping. For diffuse scenes, the shading is computed in a preprocessing step whose cost is proportional to the number of light source samples, but while the scene is static, it can be redisplayed in time independent of the number of light sources. The method is also useful for simulating the hard shadows due to a large number of point sources. The memory requirements of the algorithm are also independent of the number of light source samples.

1.1 The Idea

For diffuse scenes, our method works by precomputing, for each polygon in the scene, a *radiance texture* [12,14] that records the color (outgoing radiance) at each point in the polygon. In a diffuse scene, the radiance at each surface point is view independent, so it can be precomputed and re-used until the scene geometry changes. This radiance texture is analogous to the mesh of radiosity values computed in a radiosity algorithm. Unlike a radiosity algorithm, however, our algorithm can compute this texture almost entirely in hardware.

The key idea is to use graphics hardware to determine visibility and calculate shading, that is, to determine which portions of a surface are occluded with respect to a given extended light source, and how brightly they are lit. In order to simulate extended light sources, we approximate them with a number of *light sample points*, and we do visibility tests between a given surface point and each light sample. To keep as many operations in hardware as possible, however, we do not use a hemicube [7] to determine visibility. Instead, to compute the shadows for a single polygon, we render the scene into a scratch buffer, with all polygons except the one being shaded appropriately blackened, using a special projective projection from the point of view of each light sample. These views are registered so that corresponding pixels map to identical points on the polygon. When the resulting hard shadow images are averaged, a soft shadow image results (figure 1). This image is then used directly as a texture on the polygon in order to simulate shadows correctly. The textures so computed are used for real-time display until the scene geometry changes.

In the remainder of the paper, we summarize previous shadow algorithms, we present our method for diffuse scenes in more detail, we discuss generalizations to scenes with specular and general reflectance, we present our implementation and results, and we offer some concluding remarks.

2 Previous Work

2.1 Shadow Algorithms

Woo *et al.* surveyed a number of shadow algorithms [19]. Here we summarize soft shadows methods and methods that run at interactive rates. Shadow algorithms can be divided into three categories: those that compute everything on the fly, those that precompute just visibility, and those that precompute shading.

Computation on the Fly. Simple ray tracing computes everything on the fly. Shadows are computed on a point-by-point basis by tracing rays between the surface point and a point on each light source to check for occluders. Soft shadows can be simulated by tracing rays to a number of points distributed across the light source [8].

The shadow volume approach is another method for computing shadows on the fly. With this method, one constructs imaginary surfaces that bound the shadowed volume of space with respect to each point light source. Determining if a point is in shadow then reduces to point-in-volume testing. Brotman and Badler used an extended z-buffer algorithm with linked lists at each pixel to support soft shadows using this approach [4].

The shadow volume method has also been used in two hardware implementations. Fuchs *et al.* used the pixel processors of the Pixel Planes machine to simulate hard shadows in real-time [10]. Heidmann used the stencil buffer in advanced SGI machines [13]. With Heidmann's algorithm, the scene must be rendered through the stencil created from each light source, so the cost per frame is proportional to the number of light sources times the number of polygons. On 1991 hardware, soft shadows in a fairly simple scene required several seconds with his algorithm. His method appears to be one of the algorithms best suited to interactive use on widely available graphics hardware. We would prefer, however, an algorithm whose cost is sublinear in the number of light sources.

A simple, brute force approach, good for casting shadows of objects onto a plane, is to find the projective transformation that projects objects from a point light onto a plane, and to use it to draw each squashed, blackened object on top of the plane [3], [15, p. 401]. This algorithm effectively multiplies the number of objects in the scene by the number of light sources times the number of *receiver* polygons onto which shadows are being cast, however, so it is typically practical only for very small numbers of light sources and receivers. Another problem with this method is that occluders behind the receiver will cast erroneous shadows, unless extra clipping is done.

Precomputation of Visibility. Instead of computing visibility on the fly, one can precompute visibility from the point of view of each light source.

The z-buffer shadow algorithm uses two (or more) passes of z-buffer rendering, first from the light sources, and then from the eye [18]. The z-buffers from the light views are used in the final

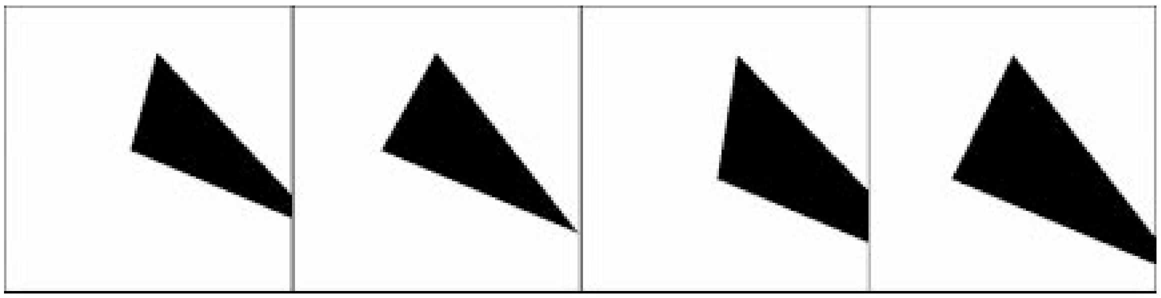


Figure 1: Hard shadow images from 2×2 grid of sample points on light source.

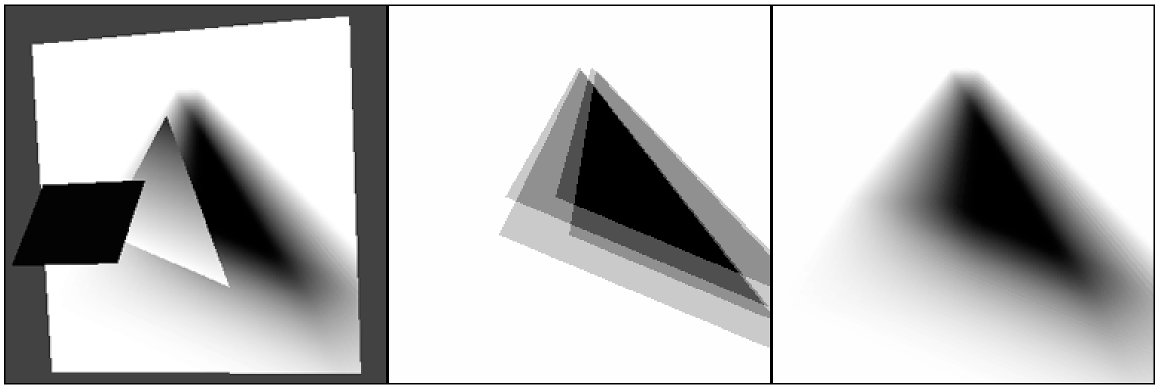


Figure 2: Left: scene with square light source (foreground), triangular occluder (center), and rectangular receiver (background), with shadows on receiver. Center: Approximate soft shadows resulting from 2×2 grid of sample points; the average of the four hard shadow images in Figure 1. Right: Correct soft shadow image (generated with 16×16 sampling). This image is used as the texture on the receiver at left.

pass to determine if a given 3-D point is illuminated with respect to each light source. The transformation of points from one coordinate system to another can be accelerated using texture mapping hardware [17]. This latter method, by Segal *et al.*, achieves real-time rates, and is the other leading method for interactive shadows. Soft shadows can be generated on a graphics workstation by rendering the scene multiple times, using different points on the extended light source, averaging the resulting images using accumulation buffer hardware [11].

A variation of the shadow volume approach is to intersect these volumes with surfaces in the scene to precompute the umbra and penumbra regions on each surface [16]. During the final rendering pass, illumination integrals are evaluated at a sparse sampling of pixels.

Precomputation of Shading. Precomputation can be taken further, computing not just visibility but also shading. This is most relevant to diffuse scenes, since their shading is view-independent. Some of these methods compute visibility continuously, while others compute it discretely.

Several researchers have explored continuous visibility methods for soft shadow computation and radiosity mesh generation. With this approach, surfaces are subdivided into fully lit, penumbra, and umbra regions by splitting along lines or curves where visibility changes. In Chin and Feiner's soft shadow method, polygons are split using BSP trees, and these sub-polygons are then pre-shaded [6]. They achieved rendering times of under a minute for simple scenes. Drettakis and Fiume used more sophisticated computational geometry techniques to precompute their subdivision, and reported rendering times of several seconds [9].

Most radiosity methods discretize each surface into a mesh of elements and then use discrete methods such as ray tracing or hemicubes to compute visibility. The hemicube method computes visibility from a light source point to an entire hemisphere by projecting the scene onto a half-cube [7]. Much of this computation can be done in hardware. Radiosity meshes typically do not resolve shadows well, however. Typical artifacts are Mach bands along the mesh element boundaries and excessively blurry shadows. Most radiosity methods are not fast enough to support interactive changes to the geometry, however. Chen's incremental radiosity method is an exception [5].

Our own method can be categorized next to hemicube radiosity methods, since it also precomputes visibility discretely. Its technique for computing visibility also has parallels to the method of flattening objects to a plane.

2.2 Graphics Hardware

Current graphics hardware, such as the Silicon Graphics Reality Engine [1], can projective-transform, clip, shade, scan convert, and texture tens of thousands of polygons in real-time (in 1/30 sec.). We would like to exploit the speed of this hardware to simulate soft shadows.

Typically, such hardware supports arbitrary 4×4 homogeneous transformations of planar polygons, clipping to any truncated pyramidal frustum (right or oblique), and scan conversion with z-buffering or overwriting. On SGI machines, Phong shading (once per pixel) is not possible, but faceted shading (once per polygon) and Gouraud shading (once per vertex) are supported. Phong shading can be simulated by splitting polygons into small pieces on input. A common, general form for hardware-supported illumination is diffuse reflection from multiple point spotlight sources, with a texture mapped reflectance function and attenuation:

$$I_c(x,y) = T_c(u,v) \sum_l \frac{\cos\theta_l \cos\theta_l'^e \, L_{lc}}{\alpha + \beta r_l + \gamma r_l^2}$$

where c is color channel index (= r, g, or b), $I_c(x,y)$ is the pixel value at screen space (x,y), $T_c(u,v)$ is a texture parameterized by texture coordinates (u,v), which are a projective transform of (x,y), θ_l is the polar angle for the ray to light source l, θ_l' is the angle away from the directional axis of the light source, e is the spotlight exponent, L_{lc} is the radiance of light l, r_l is distance to light source l, and α, β, and γ are constants controlling attenuation. Texture mapping, lights, and attenuation can be turned on and off independently on a per-polygon basis. Most systems also support Phong illumination, which has an additional specular term that we have not shown. The most advanced, expensive machines support all of these functions in hardware, while the cheaper machines do some of these calculations in software. Since the graphics subroutine interface, such as OpenGL [15], is typically identical on any machine, these differences are transparent to the user, except for the dramatic differences in running speed. So when we speak of a computation being done "in hardware", that is true only on high end machines.

The accumulation buffer [11], another feature of some graphics workstations, is hardware that allows a linear combination of images to be easily computed. It is capable of computing expressions of the general form:

$$A_c(x,y) = \sum_i \alpha_i I_{ic}(x,y)$$

where I_{ic} is a channel of image i, and A_c is a channel of the accumulator array.

3 Diffuse Scenes

Our shadow generation method for diffuse scenes takes advantage of these hardware capabilities.

Direct illumination in a scene of opaque surfaces that emit or reflect light diffusely is given by the following formula:

$$L_c(\mathbf{x}) = \rho_c(\mathbf{x}) \left(L_{\text{ac}} + \int_{\text{lights}} \frac{\cos_+\theta \cos_+\theta'}{\pi r^2} \, v(\mathbf{x},\mathbf{x}') \, L_c(\mathbf{x}') \, d\mathbf{x}' \right),$$

where, as shown in Figure 3,

- $\mathbf{x} = (x,y,z)$ is a 3-D point on a reflective surface, and $\mathbf{x}'$ is a point on a light source,
- θ is polar angle (angle from normal) at $\mathbf{x}$, θ' is the angle at $\mathbf{x}'$,
- r is the distance between $\mathbf{x}$ and $\mathbf{x}'$,
- θ, θ', and r are functions of $\mathbf{x}$ and $\mathbf{x}'$,
- $L_c(\mathbf{x})$ is outgoing radiance at point $\mathbf{x}$ for color channel c, due to either emission or reflection, L_{ac} is ambient radiance,
- $\rho_c(\mathbf{x})$ is reflectance,
- $v(\mathbf{x},\mathbf{x}')$ is a Boolean visibility function that equals 1 if point $\mathbf{x}$ is visible from point $\mathbf{x}'$, else 0,
- $\cos_+\theta = \max(\cos\theta, 0)$, for backface testing, and
- the integral is over all points on all light sources, with respect to $d\mathbf{x}'$, which is an infinitesimal area on a light source.

The inputs to the problem are the geometry, the reflectance $\rho_c(\mathbf{x})$, and emitted radiance $L_c(\mathbf{x}')$ on all light sources, the ambient radiance L_{ac}, and the output is the reflected radiance function $L_c(\mathbf{x})$.

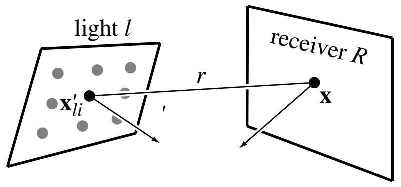


Figure 3: Geometry for direct illumination. The radiance at point $\mathbf{x}$ on the receiver is being calculated by summing the contributions from a set of point light sources at $\mathbf{x}'_{li}$ on light l.

3.1 Approximating Extended Light Sources

Although such integrals can be solved in closed form for planar surfaces with no occlusion ($v \equiv 1$), the complexity of the visibility function makes these integrals intractable in the general case. We can compute approximations to the integral, however, by replacing each extended light source l by a set of n_l point light sources:

$$L_c(\mathbf{x}') \approx \sum_l \sum_{i=1}^{n_l} a_{li} \, L_c(\mathbf{x}') \, \delta(\mathbf{x}' - \mathbf{x}'_{li}),$$

where $\delta(\mathbf{x})$ is a 3-D Dirac delta function, $\mathbf{x}'_{li}$ is sample point i on light source l, and a_{li} is the area associated with this sample point. Typically, each sample on a light source has equal area: $a_{li} = a_l/n_l$, where a_l is the area of light source l.

With this approximation, the radiance of a reflective surface point can be computed by summing the contributions over all sample points on all light sources:

$$L_c(\mathbf{x}) = \rho_c(\mathbf{x}) \, L_{\text{ac}} + \rho_c(\mathbf{x}) \sum_l \sum_{i=1}^{n_l} a_{li} \frac{\cos_+\theta_{li} \cos_+\theta'_{li}}{\pi r_{li}^2} \, v(\mathbf{x},\mathbf{x}'_{li}) \, L_c(\mathbf{x}'_{li}). \tag{1}$$

The formulas above can be generalized to linear and point light sources, as well as area light sources.

The most difficult and expensive part of the above calculation is evaluation of the visibility function v, since it requires global knowledge of the scene, whereas the remaining factors require only local knowledge. But computing v is necessary in order to simulate shadows. The above formula could be evaluated using ray tracing, but the resulting algorithm would be slow due to the large number of light source samples.

3.2 Soft Shadows in Hardware

Equation (1) can be rewritten in a form suitable to hardware computation:

$$L_c(\mathbf{x}) = \rho_c(\mathbf{x}) \, L_{\text{ac}} + \sum_l \sum_{i=1}^{n_l} \big(a_{li} \, \rho_c(\mathbf{x})\big) \left(\frac{\cos_+\theta_{li} \cos_+\theta'_{li} \, L_c(\mathbf{x}'_{li})}{\pi r_{li}^2} \right) v(\mathbf{x},\mathbf{x}'_{li}). \tag{2}$$

Each term in the inner summation can be regarded as a hard shadow image resulting from a point light source at $\mathbf{x}'_{li}$, where $\mathbf{x}$ is a function of screen (x,y).

That summand consists of the product of three factors. The first one, which is an area times the reflectance of the receiving polygon, can be calculated in software. The second factor is the cosine of the angle on the receiver, times the cosine of the angle on the light

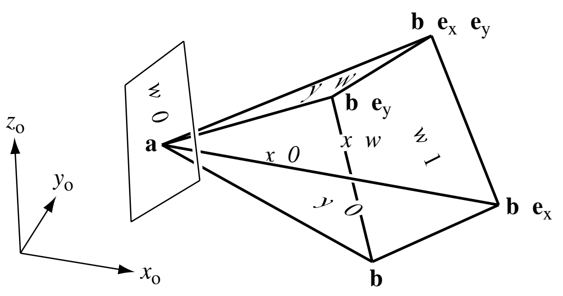


Figure 4: Pyramid with parallelogram base. Faces of pyramid are marked with their plane equations.

source, times the radiance of the light source, divided by r^2. This can be computed in hardware by rendering the receiver polygon with a single spotlight at $\mathbf{x}'_{li}$ turned on, using a spotlight exponent of $e = 1$ and quadratic attenuation. On machines that do not support Phong shading, we will have to finely subdivide the polygon. The third factor is visibility between a point on a light source and each point on the receiver. Visibility can be computed by projecting all polygons between light source point $\mathbf{x}'_{li}$ and the receiver onto the receiver.

We want to simulate soft shadows as quickly as possible. To take full advantage of the hardware, we can precompute the shading for each polygon using the formula above, and then display views of the scene from moving viewpoints using real-time texture mapping and z-buffering.

To compute soft shadow textures, we need to generate a number of hard shadow images and then average them. If these hard shadow images are not registered (they would not be, using hemi-cubes), then it would be necessary to resample them so that corresponding pixels in each hard shadow image map to the same surface point in 3-D. This would be very slow. A faster alternative is to choose the transformation for each projection so that the hard shadow images are perfectly registered with each other.

For planar receiver surfaces, this is easily accomplished by exploiting the capabilities of projective transformations. If we fit a parallelogram around the receiver surface of interest, and then construct a pyramid with this as its base and the light point as its apex, there is a 4×4 homogeneous transformation that will map such a pyramid into an axis-aligned box, as described shortly.

The hard shadow image due to sample point i on light l is created by loading this special transformation matrix and rendering the receiver polygon. The polygon is illuminated by the ambient light plus a single point light source at $\mathbf{x}'_{li}$, using Phong shading or a good approximation to it. The visibility function is then computed by rendering the remainder of the scene with all surfaces shaded as if they were the receiver illuminated by ambient light: $(r, g, b) = (\rho_r L_{ar}, \rho_g L_{ag}, \rho_b L_{ab})$. This is most quickly done with z-buffering off, and clipping to a pyramid with the receiver polygon as its base. Drawing each polygon with an unsorted painter's algorithm suffices here because all polygons are the same color, and after clipping, the only polygon fragments remaining will lie between the light source and the receiver, so they all cast shadows on the receiver. To compute the weighted average of the hard shadow images so created, we use the accumulation buffer.

3.3 Projective Transformation of a Pyramid to a Box

We want a projective (perspective) transformation that maps a pyramid with parallelogram base into a rectangular parallelepiped. The pyramid lies in object space, with coordinates (x_o, y_o, z_o). It has apex $\mathbf{a}$ and its parallelogram base has one vertex at $\mathbf{b}$ and edge vectors $\mathbf{e}_x$ and $\mathbf{e}_y$ (bold lower case denotes a 3-D point or vector). The parallelepiped lies in what we will call unit screen space, with coordinates (x_u, y_u, z_u). Viewed from the apex, the left and right sides of the pyramid map to the parallel planes $x_u = 0$ and $x_u = 1$, the bottom and top map to $y_u = 0$ and $y_u = 1$, and the base plane and a plane parallel to it through the apex map to $z_u = 1$ and $z_u = \infty$, respectively. See figure 4.

A 4×4 homogeneous matrix effecting this transformation can be derived from these conditions. It will have the form:

$$\mathbf{M} = \begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ 0 & 0 & 0 & 1 \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix},$$

and the homogeneous transformation and homogeneous division to transform object space to unit screen space are:

$$\begin{pmatrix} x \\ y \\ 1 \\ w \end{pmatrix} = \mathbf{M} \begin{pmatrix} x_o \\ y_o \\ z_o \\ 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} x_u \\ y_u \\ z_u \end{pmatrix} = \begin{pmatrix} x/w \\ y/w \\ 1/w \end{pmatrix}.$$

The third row of matrix $\mathbf{M}$ takes this simple form because a constant z_u value is desired on the base plane. The homogeneous screen coordinates x, y, and w are each affine functions of x_o, y_o, and z_o (that is, linear plus translation). The constraints above specify the value of each of the three coordinates at four points in space – just enough to uniquely determine the twelve unknowns in $\mathbf{M}$.

The w coordinate, for example, has value 1 at the points $\mathbf{b}$, $\mathbf{b}+\mathbf{e}_x$, and $\mathbf{b}+\mathbf{e}_y$, and value 0 at $\mathbf{a}$. Therefore, the vector $\mathbf{n}_w = \mathbf{e}_y \times \mathbf{e}_x$ is normal to any plane of constant w, thus fixing the first three elements of the last row of the matrix within a scale factor: $(m_{30}, m_{31}, m_{32})^T = \alpha_w \mathbf{n}_w$. Setting w to 0 at $\mathbf{a}$ and 1 at $\mathbf{b}$ constrains $m_{33} = -\alpha_w \mathbf{n}_w \cdot \mathbf{a}$ and $\alpha_w = 1/\mathbf{n}_w \cdot \mathbf{e}_w$, where $\mathbf{e}_w = \mathbf{b} - \mathbf{a}$. The first two rows of $\mathbf{M}$ can be derived similarly (see figure 4). The result is:

$$\mathbf{M} = \begin{pmatrix} \alpha_x n_{xx} & \alpha_x n_{xy} & \alpha_x n_{xz} & -\alpha_x \mathbf{n}_x \cdot \mathbf{b} \\ \alpha_y n_{yx} & \alpha_y n_{yy} & \alpha_y n_{yz} & -\alpha_y \mathbf{n}_y \cdot \mathbf{b} \\ 0 & 0 & 0 & 1 \\ \alpha_w n_{wx} & \alpha_w n_{wy} & \alpha_w n_{wz} & -\alpha_w \mathbf{n}_w \cdot \mathbf{a} \end{pmatrix},$$

where

$$\begin{aligned} \mathbf{n}_x &= \mathbf{e}_w \times \mathbf{e}_y \\ \mathbf{n}_y &= \mathbf{e}_x \times \mathbf{e}_w \\ \mathbf{n}_w &= \mathbf{e}_y \times \mathbf{e}_x \end{aligned} \quad \text{and} \quad \begin{aligned} \alpha_x &= 1/\mathbf{n}_x \cdot \mathbf{e}_x \\ \alpha_y &= 1/\mathbf{n}_y \cdot \mathbf{e}_y \\ \alpha_w &= 1/\mathbf{n}_w \cdot \mathbf{e}_w \end{aligned}.$$

Blinn [3] uses a related projective transformation for the generation of shadows on a plane, but his is a projection (it collapses 3-D to 2-D), while ours is 3-D to 3-D. We use the third dimension for clipping.

3.4 Using the Transformation

To use this transformation in our shadow algorithm, we first fit a parallelogram around the receiver polygon. If the receiver is a rectangle or other parallelogram, the fit is exact; if the receiver is a triangle, then we fit the triangle into the lower left triangle of the parallelogram; and for more general polygons with four or more sides, a simple 2-D bounding box in the plane of the polygon can be used. It is possible to go further with projective transformations, mapping arbitrary planar quadrilaterals into squares (using the homogeneous texture transformation matrix of OpenGL, for example). We assume for simplicity, however, that the transformation between texture space (the screen space in these light source projections) and object space is affine, and so we restrict ourselves to parallelograms.

3.5 Soft Shadow Algorithm for Diffuse Scenes

To precompute soft shadow radiance textures:

```
turn off z-buffering
for each receiver polygon $R$
  choose resolution for receiver's texture ($s_x \times s_y$ pixels)
  clear accumulator image of $s_x \times s_y$ pixels to black
  create temporary image of $s_x \times s_y$ pixels
  for each light source $l$
    first backface test: if $l$ is entirely behind $R$
      or $R$ is entirely behind $l$, then skip to next $l$
    for each sample point $i$ on light source $l$
      second backface test: if $\mathbf{x}'_{li}$ is behind $R$ then skip to next $i$
      compute transformation matrix $\mathbf{M}$, where $\mathbf{a} = \mathbf{x}'_{li}$,
        and the base parallelogram fits tightly around $R$
      set current transformation matrix to scale$(s_x, s_y, 1) \cdot \mathbf{M}$
      set clipping planes to $z_{u,near} = 1 - \epsilon$ and $z_{u,far} = $big
      draw $R$ with illumination from $\mathbf{x}'_{li}$ only, as described in
        equation (2), into temp image
      for each other object in scene
        draw object with ambient color into temp image
      add temp image into accumulator image with weight $a_l / n_l$
  save accumulator image as texture for polygon $R$
```

A hard shadow image is computed in each iteration of the i loop. These are averaged together to compute a soft shadow image, which is used as a radiance texture. Note that objects casting shadows need not be polygonal; any object that can be quickly scan converted will work well.

To display a static scene from moving viewpoints, simply:

```
turn on z-buffering
for each object in scene
  if object receives shadows, draw it textured but without illumination
  else draw object with illumination
```

3.6 Backface Testing

The cases where $\cos_+\theta \cos_+\theta' = 0$ can be optimized using backface testing.

To test if polygon p is behind polygon q, compute the signed distances from the plane of polygon q to each of the vertices of p (signed positive on the front of q and negative on the back). If they are all positive, then p is entirely in front of q, if they are all nonpositive, p is entirely in back, otherwise, part of p is in front of q and part is in back.

To test if the apex $\mathbf{a}$ of the pyramid is behind the receiver R that defines the base plane, simply test if $\mathbf{n}_w \cdot \mathbf{e}_w \leq 0$.

The above checks will ensure that $\cos\theta > 0$ at every point on the receiver, but there is still the possibility that $\cos\theta' \leq 0$ on portions of the receiver (i.e. that the receiver is only partially illuminated by the light source). This final case should be handled at the polygon level or pixel level when shading the receiver in the algorithm above. Phong shading, or a good approximation to it, is needed here.

3.7 Sampling Extended Light Sources

The set of samples used on each light source greatly influences the speed and quality of the results. Too few samples, or a poorly chosen sample distribution, result in penumbras that appear stepped, not continuous. If too many samples are used, however, the simulation runs too slowly.

If a uniform grid of sample points is used, the stepping is much more pronounced in some cases. For example, if a uniform grid of $m \times m$ samples is used on a parallelogram light source, an occluder edge coplanar with one of the light source edges will cause m big steps, while an occluder edge in general position will cause m^2 small steps.

Stochastic sampling [8] with the same number of samples yields smoother penumbra than a uniform grid, because the steps no longer coincide. We use a jittered uniform grid because it gives good results and is very easy to compute.

Using a fixed number of samples on each light source is inefficient. Fine sampling of a light source is most important when the light source subtends a large solid angle from the point of view of the receiver, since that is when the penumbra is widest and stepping artifacts would be most visible. A good approach is to choose the light source sample resolution such that the solid angle subtended by the light source area associated with each sample is below a user-specified threshold.

The algorithm can easily handle diffuse (non-directional) light sources whose outgoing radiance varies with position, such as stained glass windows. For such light sources, importance sampling might be preferable: concentration of samples in the regions of the light source with highest radiance.

3.8 Texture Resolution

The resolution of the shadow texture should be roughly equal to the resolution at which it will be viewed (one texture pixel mapping to one screen pixel); lower resolution results in visible artifacts such as blocky shadows, and higher resolution is wasteful of time and memory. In the absence of information about probable views, a reasonable technique is to set the number of pixels on a polygon's texture, in each dimension, proportional to its size in world space using a "desired pixel size" parameter. With this scheme, the required texture memory, in pixels, will be the total world space surface area of all polygons in the scene divided by the square of the desired pixel size.

Texture memory for triangles can be further optimized by packing the textures for two triangles into one rectangular texture block.

If there are too many polygons in the scene, or the desired pixel size is too small, the texture memory could be exceeded, causing paging of texture memory and slow performance.

Radiance textures can be antialiased by supersampling: generating the hard and initial soft shadow images at several times the desired resolution, and then filtering and downsampling the images before creating textures. Textured surfaces should be rendered with good texture filtering.

Some polygons will contain penumbral regions with respect to a light source, and will require high texture resolution, but others will be either totally shadowed (umbral) or totally illuminated by each light source, and will have very smooth radiance functions. Sometimes these functions will be so smooth that they can be adequately approximated by a single Gouraud shaded polygon. This optimization saves significant texture memory and speeds display.

This idea can be carried further, replacing the textured planar polygon with a mesh of coplanar Gouraud shaded triangles. For complex shadow patterns and radiance functions, however, textures may render faster than the corresponding Gouraud approximation, depending on the relative speed of texture mapping and Gouraud-shaded triangle drawing, and the number of triangles required to achieve a good approximation.

3.9 Complexity

We now analyze the expected complexity of our algorithm (worst case costs are not likely to be observed in practice, so we do not discuss them here). Although more sophisticated schemes are possible, we will assume for the purposes of analysis that the same set

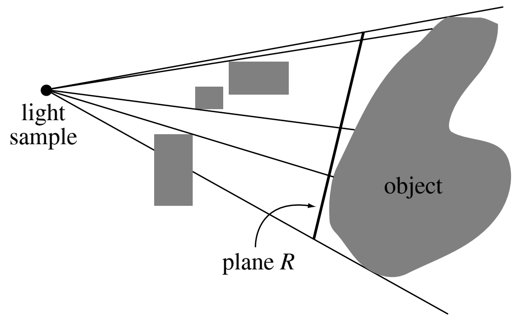


Figure 5: Shadows are computed on plane R and projected onto the receiving object at right.

of light samples are used for shadowing all polygons. Suppose we have a scene with s surfaces (polygons), a total of $n = \sum_l n_l$ light source samples, a total of t radiance texture pixels, and the output images are rendered with p pixels. We assume the depth complexity of the scene (the average number of surfaces intersecting a ray) is bounded, and that t and p are roughly linearly related. The average number of texture pixels per polygon is t/s.

With our technique, preprocessing renders the scene ns times. A painter's algorithm rendering of s polygons into an image of t/s pixels takes $O(s+t/s)$ time for scenes of bounded depth complexity. The total preprocessing time is thus $O(ns^2+nt)$, and the required texture memory is $O(t)$. Display requires only z-buffered texture mapping of s polygons to an image of p pixels, for a time cost of $O(s+p)$. The memory for the z-buffer and output image is $O(p)=O(t)$.

Our display algorithm is very fast for complex scenes. Its cost is independent of the number of light source samples used, and also independent of the number of texture pixels (assuming no texture paging).

For scenes of low or moderate complexity, our preprocessing algorithm is fast because all of its pixel operations can be done in hardware. For very complex scenes, our preprocessing algorithm becomes impractical because it is quadratic in s, however. In such cases, performance can be improved by calculating shadows only on a small number of surfaces in the scene (e.g. floor, walls, and other large, important surfaces), thereby reducing the cost to $O(nss_t+nt)$, where s_t is the number of textured polygons.

In an interactive setting, a progressive refinement of images can be used, in which hard shadows on a small number of polygons (precomputation with $n=1$, s_t small) are rendered while the user is moving objects with the mouse, a full solution (precomputation with n large, s_t large) is computed when they complete a movement, and then top speed rendering (display with texture mapping) is used as the viewer moves through the scene.

More fundamentally, the quadratic cost can be reduced using more intelligent data structures. Because the angle of view of most of the shadow projection pyramids is narrow, only a small fraction of the polygons in a scene shadow a given polygon, on average. Using spatial data structures, entire objects can be culled with a few quick tests [2], obviating transformation and clipping of most of the scene, speeding the rendering of each hard shadow image from $O(s+t/s)$ to $O(s^\alpha+t/s)$, where $\alpha \approx .3$ or so.

An alternative optimization, which would make the algorithm more practical for the generation of shadows on complex curved or many-faceted objects, is to approximate a receiving object with a plane, compute shadows on this plane, and then project the shadows onto the object (figure 5). This has the advantage of replacing many renderings with a single rendering, but its disadvantage is that self-shadowing of concave objects is not simulated.

3.10 Comparison to Other Algorithms

We can compare the complexity of our algorithm to other algorithms capable of simulating soft shadows at near-interactive rates. The main alternatives are the stencil buffer technique by Heidmann, the z-buffer method by Segal *et al.*, and hardware hemicube-based radiosity algorithms.

The stencil buffer technique renders the scene once for each light source, so its cost per frame is $O(ns+np)$, making it difficult to support soft shadows in real-time. With the z-buffer shadow algorithm, the preprocessing time is acceptable, but the memory cost and display time cost are $O(np)$. This makes the algorithm awkward for many point light sources or extended light sources with many samples (large n). When soft shadows are desired, our approach appears to yield faster walkthroughs than either of these two methods, because our display process is so fast.

Among current radiosity algorithms, progressive radiosity using hardware hemicubes is probably the fastest method for complex scenes. With progressive radiosity, very high resolution hemicubes and many elements are needed to get good shadows, however. While progressive radiosity may be a better approach for shadow generation in very complex scenes (very large s), it appears slower than our technique for scenes of moderate complexity because every pixel-level operation in our algorithm can be done in hardware, but this is not the case with hemicubes, since the process of summing differential form factors while reading out of the hemicube must be done in software [7].

4 Scenes with General Reflectance

Shadows on specular surfaces, or surfaces with more general reflectance, can be simulated with a generalization of the diffuse algorithm, but not without added time and memory costs.

Shadows from a single point light source are easily simulated by placing just the visibility function $v(\mathbf{x}, \mathbf{x}')$ in texture memory, creating a Boolean *shadow texture*, and computing the remaining local illumination factors at vertices only. This method costs $O(ss_t+t)$ for precomputation, and $O(s+p)$ for display.

Shadows from multiple point light sources can also be simulated. After precomputing a shadow texture for each polygon when illuminated with each light source, the total illumination due to n light sources can be calculated by rendering the scene n times with each of these sets of shadow textures, compositing the final image using blending or with the accumulation buffer. The cost of this method is nt one-bit texture pixels and $O(ns+np)$ display time.

Generalizing this method to extended light sources in the case of general reflectance is more difficult, as the computation involves the integration of light from polygonal light sources weighted by the bidirectional reflectance distribution functions (BRDFs). Specular BRDF's are spiky, so careful integration is required or the highlights will betray the point sampling of the light sources. We believe, however, that with careful light sampling and numerical integration of the BRDF's, soft shadows on surfaces with general reflectance could be displayed with $O(nt)$ memory and $O(ns+np)$ time.

5 Implementation

We implemented our diffuse algorithm using the OpenGL subroutine library, running with the IRIX 5.3 operating system on an SGI Crimson with 100 MHz MIPS R4000 processor and Reality Engine graphics. This machine has hardware for texture mapping and an accumulation buffer with 24 bits per channel.

The implementation is fairly simple, since OpenGL supports loading of arbitrary 4×4 matrices, and we intentionally cast our shading formulas in a form that maps cleanly into OpenGL's model. The source code is about 2,000 lines of C++. Our implementation renders at about 900×900 resolution, and uses 24-bit textures at sizes of $2^{k_x} \times 2^{k_y}$ pixels, for $2 \le k_x, k_y \le 8$. Phong shading is simulated by subdividing each receiver polygon into a grid of 8×8-pixel parallelograms during preprocessing.

Our software allows interactive movement of objects and the camera. When the scene geometry is changed, textures are recomputed. On a scene with $s = 749$ polygons, $s_t = 3$ of them textured, with two area light sources sampled with $n = 8$ points total, generating textures with about $t = 200,000$ pixels total, and a final picture of about $p = 810,000$ pixels, preprocessing has a redisplay rate of 2 Hz. For simple scenes, the slowest part of preprocessing is the transfer of radiance textures from system memory to texture memory.

When only the view is changed, we simply redisplay the scene with texture mapping. The use of OpenGL display lists helps us achieve 30 Hz rates in most cases. When we allocate more radiance texture memory than the hardware can hold, however, paging slows redisplay.

Since we know the size and perceptual importance of each object at modeling time, we have found it convenient to have each receiver object control the number of light source samples that are used to illuminate it. The floor and walls, for example, might specify many light source samples, while table and chairs might specify a single light source sample. To facilitate further testing of shadow sampling, a slider that acts as a multiplier on the requested number of samples per light source is provided. More automatic and intelligent light sampling schemes are certainly possible.

6 Results

The color figures illustrate high quality results achievable in a few seconds with fine light source sampling. Figure 6 shows a scene with 6,142 polygons, 3 of them shadowed, which was computed in 5.5 seconds using $n = 32$ light samples total on two light sources. Figure 7 illustrates the calculation of shadows on more complex objects, with a total of $s_t = 25$ shadowed polygons. For this image, 7×7 light sampling was used when shadowing the walls and floor, while 3×3 sampling was used to compute shadows on the table top, and 2×2 sampling was used for the table legs. The textures for the table polygons are smaller than those for the walls and floor, in proportion to their world space size. This image was calculated in 13 seconds.

7 Conclusions

We have described a simple algorithm for generating soft shadows at interactive rates by exploiting graphics workstation hardware. Previous shadow generation methods have not supported both the computation and display of soft shadows at these speeds.

To achieve real time rates with our method, one probably needs hardware support for transformation, clipping, scan conversion, texture mapping, and accumulation buffer operations. In coming years, such hardware will only become more affordable, however. Software implementations will also work, of course, but at reduced speeds.

For most scenes, realistic images can be generated by computing soft shadows only for a small set of polygons. This will run quite fast. If it is necessary to compute shadows for every polygon, our preprocessing method has quadratic growth with respect to scene complexity s, but we believe this can be reduced to about $O(s^{1.3})$, using spatial data structures to cull off-screen objects.

Once preprocessing is done, the display cost is independent of the number and size of light sources. This cost is little more than the display cost without shadows.

The method also has potential as a form factor calculation technique for progressive radiosity.

8 Acknowledgments & Notes

We thank Silicon Graphics for the gift of a Reality Engine, which made this work possible. Jeremiah Blatz and Michael Garland provided modeling assistance. This paper grew out of a project by Herf in a graduate course on Rendering taught by Heckbert, Fall 1995.

References

[1] Kurt Akeley. RealityEngine graphics. In *SIGGRAPH '93 Proc.*, pages 109–116, Aug. 1993.

[2] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An introduction to ray tracing*, pages 201–262. Academic Press, 1989.

[3] James F. Blinn. Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, 8(1):82–86, Jan. 1988.

[4] Lynne Shapiro Brotman and Norman I. Badler. Generating soft shadows with a depth buffer algorithm. *IEEE Computer Graphics and Applications*, 4(10):5–24, Oct. 1984.

[5] Shenchang Eric Chen. Incremental radiosity: An extension of progressive radiosity to an interactive image synthesis system. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):135–144, August 1990.

[6] Norman Chin and Steven Feiner. Fast object-precision shadow generation for area light sources using BSP trees. In *1992 Symp. on Interactive 3D Graphics*, pages 21–30. ACM SIGGRAPH, Mar. 1992.

[7] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: A radiosity solution for complex environments. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):31–40, July 1985.

[8] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. on Graphics*, 5(1):51–72, Jan. 1986.

[9] George Drettakis and Eugene Fiume. A fast shadow algorithm for area light sources using backprojection. In *SIGGRAPH '94 Proc.*, pages 223–230, 1994. http://safran.imag.fr/Membres/George.Drettakis/pub.html.

[10] Henry Fuchs, Jack Goldfeather, Jeff P. Hultquist, Susan Spach, John D. Austin, Frederick P. Brooks, Jr., John G. Eyles, and John Poulton. Fast spheres, shadows, textures, transparencies, and image enhancements in Pixel-Planes. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):111–120, July 1985.

[11] Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):309–318, Aug. 1990.

[12] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):145–154, Aug. 1990.

[13] Tim Heidmann. Real shadows, real time. *Iris Universe*, 18:28–31, 1991. Silicon Graphics, Inc.

[14] Karol Myszkowski and Tosiyasu L. Kunii. Texture mapping as an alternative for meshing during walkthrough animation. In *Fifth Eurographics Workshop on Rendering*, pages 375–388, June 1994.

[15] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 1993.

[16] Tomoyuki Nishita and Eihachiro Nakamae. Half-tone representation of 3-D objects illuminated by area sources or polyhedron sources. In *COMPSAC '83, Proc. IEEE 7th Intl. Comp. Soft. and Applications Conf.*, pages 237–242, Nov. 1983.

[17] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):249–252, July 1992.

[18] Lance Williams. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, 12(3):270–274, Aug. 1978.

[19] Andrew Woo, Pierre Poulin, and Alain Fournier. A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, 10(6):13–32, Nov. 1990.

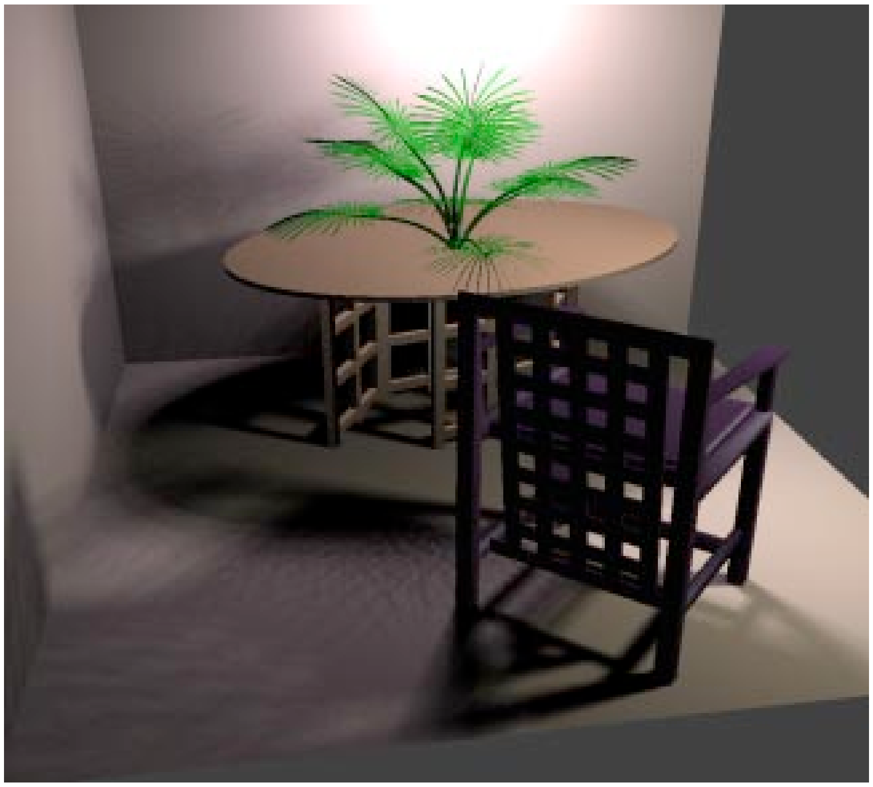


Figure 6: Shadows on walls and floor, computed in 5.5 seconds.

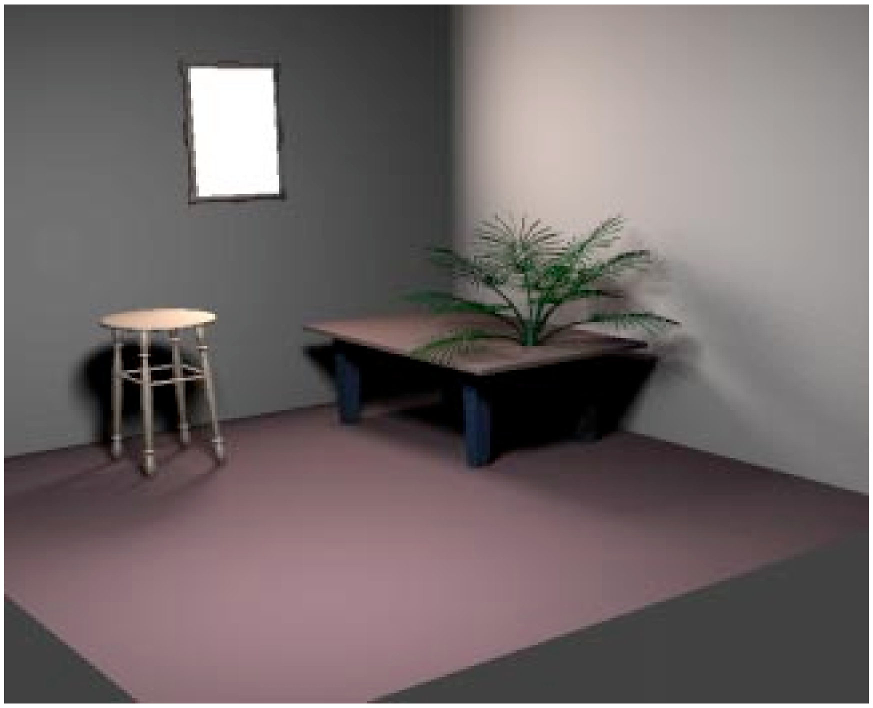


Figure 7: Shadows on walls, floor, and table, computed in 13 seconds.

A Non-Photorealistic Lighting Model For Automatic Technical Illustration

Amy Gooch **Bruce Gooch** **Peter Shirley** **Elaine Cohen**

Department of Computer Science
University of Utah
http://www.cs.utah.edu/

Abstract

Phong-shaded 3D imagery does not provide geometric information of the same richness as human-drawn technical illustrations. A non-photorealistic lighting model is presented that attempts to narrow this gap. The model is based on practice in traditional technical illustration, where the lighting model uses both luminance and changes in hue to indicate surface orientation, reserving extreme lights and darks for edge lines and highlights. The lighting model allows shading to occur only in mid-tones so that edge lines and highlights remain visually prominent. In addition, we show how this lighting model is modified when portraying models of metal objects. These illustration methods give a clearer picture of shape, structure, and material composition than traditional computer graphics methods.

CR Categories: I.3.0 [Computer Graphics]: General; I.3.6 [Computer Graphics]: Methodology and Techniques.

Keywords: illustration, non-photorealistic rendering, silhouettes, lighting models, tone, color, shading

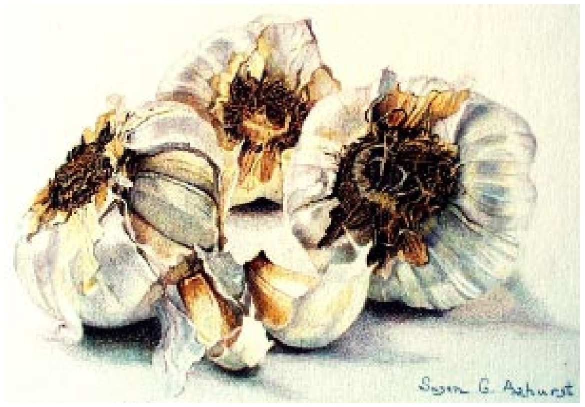


Figure 1: *The non-photorealistic cool (blue) to warm (tan) transition on the skin of the garlic in this non-technical setting is an example of the technique automated in this paper for technical illustrations. Colored pencil drawing by Susan Ashurst.*

1 Introduction

The advent of photography and computers has not replaced artists, illustrators, or draftsmen, despite rising salaries and the decreasing cost of photographic and computer rendering technology. Almost all manuals that involve 3D objects, e.g., a car owner's manual, have illustrations rather than photographs. This lack of photography is present even in applications where aesthetics are a side-issue, and communication of geometry is the key. Examining technical manuals, illustrated textbooks, and encyclopedias reveals illustration conventions that are quite different from current computer graphics methods. These conventions fall under the umbrella term *technical illustrations*. In this paper we attempt to automate some of these conventions. In particular, we adopt a shading algorithm based on cool-to-warm *tones* such as shown in the non-technical image in Figure 1. We adopt this style of shading to ensure that black silhouettes and edge lines are clearly visible which is often not the case when they are drawn in conjunction with traditional computer graphics shading. The fundamental idea in this paper is that when silhouettes and other edge lines are explicitly drawn, then very low dynamic range shading is needed for the interior. As artists have discovered, adding a somewhat artificial hue shift to shading helps imply shape without requiring a large dynamic range. This hue shift can interfere with precise albedo perception, but this is not a major concern in technical illustration where the communication of shape and form are valued above realism. In Section 2 we review previous computer graphics work, and conclude that little has been done to produce shaded technical drawings. In Section 3 we review the common technical illustration practices. In Section 4 we describe how we have automated some of these practices. We discuss future work and summarize in Section 5.

2 Related Work

Computer graphics algorithms that imitate non-photographic techniques such as painting or pen-and-ink are referred to as *non-photorealistic rendering* (NPR). The various NPR methods differ greatly in style and visual appearance, but are all closely related to conventional artistic techniques (e.g., [6, 8, 10, 13, 14, 16, 20, 26]). An underlying assumption in NPR is that artistic techniques developed by human artists have intrinsic merit based on the evolutionary nature of art. We follow this assumption in the case of technical illustration.

NPR techniques used in computer graphics vary greatly in their level of abstraction. Those that produce a loss of detail, such as semi-randomized watercolor or pen-and-ink, produce a very high level of abstraction, which would be inappropriate for most technical illustrations. Photorealistic rendering techniques provide little abstraction, so photorealistic images tend to be more confusing than less detailed human-drawn technical illustrations. Technical illustrations occupy the middle ground of abstraction, where the important three-dimensional properties of objects are accented while extraneous detail is diminished or eliminated. Images at any level of abstraction can be aesthetically pleasing, but this is a side-effect rather than a primary goal for technical illustration. A rationale for using abstraction to eliminate detail from an image is that, unlike the case of 3D scene perception, the image viewer is not able to use motion, accommodation, or parallax cues to help deal with visual complexity. Using abstraction to simplify images helps the user overcome the loss of these spatial cues in a 2D image.

In computer graphics, there has been little work related to technical illustration. Saito and Takahashi [19] use a variety of techniques to show geometric properties of objects, but their images do not follow many of the technical illustration conventions. Seligmann and Feiner present a system that automatically generates explanation-based drawings [21]. Their system focuses primarily on what to draw, with secondary attention to visual style. Our work deals primarily with visual style rather than layout issues, and thus there is little overlap with Seligmann and Feiner's system, although the two methods would combine naturally. The work closest to our own was presented by Dooley and Cohen [7] who employ a user-defined hierarchy of components, such as line width, transparency, and line end/boundary conditions to generate an image. Our goal is a simpler and more automatic system, that imitates methods for line and color use found in technical illustrations. Williams also developed similar techniques to those described here for non-technical applications, including some warm-to-cool tones to approximate global illumination, and drawing conventions for specular objects [25].

3 Illustration Techniques

Based on the illustrations in several books, e.g. [15, 18], we conclude that illustrators use fairly algorithmic principles. Although there are a wide variety of styles and techniques found in technical illustration, there are some common themes. This is particularly true when we examine color illustrations done with air-brush and pen. We have observed the following characteristics in many illustrations:

- edge lines, the set containing surface boundaries, silhouettes, and discontinuities, are drawn with black curves.
- matte objects are shaded with intensities far from black or white with warmth or coolness of color indicative of surface normal; a single light source provides white highlights.
- shadowing is not shown.
- metal objects are shaded as if very anisotropic.

We view these characteristics as resulting from a hierarchy of priorities. The edge lines and highlights are black and white, and provide a great deal of shape information themselves. Several studies in the field of perception have concluded that subjects can recognize 3D objects at least as well, if not better, when the edge lines (contours) are drawn versus shaded or textured images [1, 3, 5, 22]. However, when shading is added in addition to edge lines, more information is provided only if the shading uses colors that are visually distinct from both black and white. This means the dynamic range available for shading is extremely limited. In most technical illustrations, shape information is valued above precise reflectance information, so hue changes are used to indicate surface orientation rather than reflectance. This theme will be investigated in detail in the next section.

A simple low dynamic-range shading model is consistent with several of the principles from Tufte's recent book [23]. He has a case-study of improving a computer graphics animation by lowering the contrast of the shading and adding black lines to indicate direction. He states that this is an example of the strategy of *the smallest effective difference*:

Make all visual distinctions as subtle as possible, but still clear and effective.

Tufte feels that this principle is so important that he devotes an entire chapter to it. The principle provides a possible explanation of why cross-hatching is common in black and white drawings and rare in colored drawings: colored shading provides a more subtle, but adequately effective, difference to communicate surface orientation.

4 Automatic Lighting Model

All of the characteristics from Section 3 can be automated in a straightforward manner. Edge lines are drawn in black, and highlights are drawn using the traditional exponential term from the Phong lighting model [17]. In Section 4.1, we consider matte objects and present reasons why traditional shading techniques are insufficient for technical illustration. We then describe a low dynamic range artistic tone algorithm in Section 4.2. Next we provide an alogrithm to approximate the anisotropic appearance of metal objects, described in Section 4.3. We provide approximations to these algorithms using traditional Phong shading in Section 4.4.

4.1 Traditional Shading of Matte Objects

In addition to drawing edge lines and highlights, we need to shade the surfaces of objects. Traditional diffuse shading sets luminance proportional to the cosine of the angle between light direction and surface normal:

$$I = k_d k_a + k_d \ \max\left(0, \hat{\mathbf{l}} \cdot \hat{\mathbf{n}}\right) \tag{1}$$

where I is the RGB color to be displayed for a given point on the surface, k_d is the RGB diffuse reflectance at the point, k_a is the RGB ambient illumination, $\hat{\mathbf{l}}$ is the unit vector in the direction of the light source, and $\hat{\mathbf{n}}$ is the unit surface normal vector at the point. This model is shown for $k_d = 1$ and $k_a = 0$ in Figure 3. This unsatisfactory image hides shape and material information in the dark regions. Additional information about the object can be provided by both highlights and edge lines. These are shown alone in Figure 4 with no shading. We cannot effectively add edge lines and highlights to Figure 3 because the highlights would be lost in the light regions and the edge lines would be lost in the dark regions.

To add edge lines to the shading in Equation 1, we can use either of two standard heuristics. First we could raise k_a until it is large enough that the dim shading is visually distinct from the black edge lines, but this would result in loss of fine details. Alternatively, we could add a second light source, which would add conflicting highlights and shading. To make the highlights visible on top of the shading, we can lower k_d until it is visually distinct from white. An image with hand-tuned k_a and k_d is shown in Figure 5. This is the best achromatic image using one light source and traditional shading. This image is poor at communicating shape information, such as details in the claw nearest the bottom of the image. This part of the image is colored the constant shade $k_d k_a$ regardless of surface orientation.

4.2 Tone-based Shading of Matte Objects

In a colored medium such as air-brush and pen, artists often use both hue and luminance (greyscale intensity) shifts. Adding blacks and whites to a given color results in what artists call *shades* in the case of black, and *tints* in the case of white. When color scales are

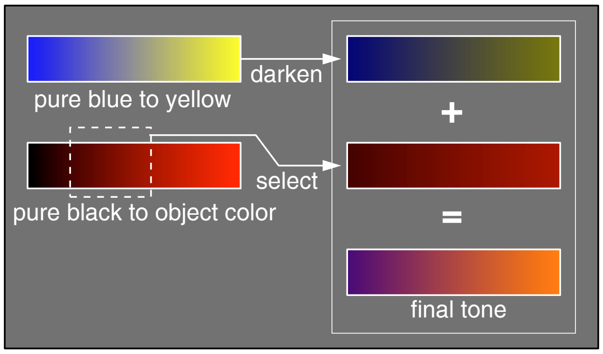


Figure 2: *How the tone is created for a pure red object by summing a blue-to-yellow and a dark-red-to-red tone.*

created by adding grey to a certain color they are called *tones* [2]. Such tones vary in hue but do not typically vary much in luminance. When the complement of a color is used to create a color scale, they are also called tones. Tones are considered a crucial concept to illustrators, and are especially useful when the illustrator is restricted to a small luminance range [12]. Another quality of color used by artists is the *temperature* of the color. The temperature of a color is defined as being warm (red, orange, and yellow), cool (blue, violet, and green), or temperate (red-violets and yellow-greens). The depth cue comes from the perception that cool colors recede while warm colors advance. In addition, object colors change temperature in sunlit scenes because cool skylight and warm sunlight vary in relative contribution across the surface, so there may be ecological reasons to expect humans to be sensitive to color temperature variation. Not only is the temperature of a hue dependent upon the hue itself, but this advancing and receding relationship is effected by proximity [4]. We will use these techniques and their psychophysical relationship as the basis for our model.

We can generalize the classic computer graphics shading model to experiment with tones by using the cosine term ($\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}$) of Equation 1 to blend between two RGB colors, k_{cool} and k_{warm}:

$$I = \left(\frac{1 + \hat{\mathbf{l}} \cdot \hat{\mathbf{n}}}{2}\right) k_{cool} + \left(1 - \frac{1 + \hat{\mathbf{l}} \cdot \hat{\mathbf{n}}}{2}\right) k_{warm} \qquad (2)$$

Note that the quantity $\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}$ varies over the interval $[-1, 1]$. To ensure the image shows this full variation, the light vector $\hat{\mathbf{l}}$ should be perpendicular to the gaze direction. Because the human vision system assumes illumination comes from above [9], we chose to position the light up and to the right and to keep this position constant.

An image that uses a color scale with little luminance variation is shown in Figure 6. This image shows that a sense of depth can be communicated at least partially by a hue shift. However, the lack of a strong cool to warm hue shift and the lack of a luminance shift makes the shape information subtle. We speculate that the unnatural colors are also problematic.

In order to automate this hue shift technique and to add some luminance variation to our use of tones, we can examine two extreme possibilities for color scale generation: blue to yellow tones and scaled object-color shades. Our final model is a linear combination of these techniques. Blue and yellow tones are chosen to insure a cool to warm color transition regardless of the diffuse color of the object.

The blue-to-yellow tones range from a fully saturated blue: $k_{blue} = (0, 0, b), b \in [0, 1]$ in RGB space to a fully saturated yellow: $k_{yellow} = (y, y, 0), y \in [0, 1]$. This produces a very sculpted but unnatural image, and is independent of the object's diffuse reflectance k_d. The extreme tone related to k_d is a variation of diffuse shading where k_{cool} is pure black and $k_{warm} = k_d$. This would look much like traditional diffuse shading, but the entire object would vary in luminance, including where $\hat{\mathbf{l}} \cdot \hat{\mathbf{n}} < 0$. What we would really like is a compromise between these strategies. These transitions will result in a combination of tone scaled object-color and a cool-to-warm undertone, an effect which artists achieve by combining pigments. We can simulate undertones by a linear blend between the blue/yellow and black/object-color tones:

$$\begin{aligned} k_{cool} &= k_{blue} + \alpha k_d \\ k_{warm} &= k_{yellow} + \beta k_d \end{aligned} \qquad (3)$$

Plugging these values into Equation 2 leaves us with four free parameters: b, y, α, and β. The values for b and y will determine the strength of the overall temperature shift, and the values of α and β will determine the prominence of the object color and the strength of the luminance shift. Because we want to stay away from shading which will visually interfere with black and white, we should supply intermediate values for these constants. An example of a resulting tone for a pure red object is shown in Figure 2.

Substituting the values for k_{cool} and k_{warm} from Equation 3 into the tone Equation 2 results in shading with values within the middle luminance range as desired. Figure 7 is shown with $b = 0.4$, $y = 0.4$, $\alpha = 0.2$, and $\beta = 0.6$. To show that the exact values are not crucial to appropriate appearance, the same model is shown in Figure 8 with $b = 0.55$, $y = 0.3$, $\alpha = 0.25$, and $\beta = 0.5$. Unlike Figure 5, subtleties of shape in the claws are visible in Figures 7 and 8.

The model is appropriate for a range of object colors. Both traditional shading and the new tone-based shading are applied to a set of spheres in Figure 9. Note that with the new shading method objects retain their "color name" so colors can still be used to differentiate objects like countries on a political map, but the intensities used do not interfere with the clear perception of black edge lines and white highlights.

4.3 Shading of Metal Objects

Illustrators use a different technique to communicate whether or not an object is made of metal. In practice illustrators represent a metallic surface by alternating dark and light bands. This technique is the artistic representation of real effects that can be seen on milled metal parts, such as those found on cars or appliances. Milling creates what is known as "anisotropic reflection." Lines are streaked in the direction of the axis of minimum curvature, parallel to the milling axis. Interestingly, this visual convention is used even for smooth metal objects [15, 18]. This convention emphasizes that realism is not the primary goal of technical illustration.

To simulate a milled object, we map a set of twenty stripes of varying intensity along the parametric axis of maximum curvature. The stripes are random intensities between 0.0 and 0.5 with the stripe closest to the light source direction overwritten with white. Between the stripe centers the colors are linearly interpolated. An object is shown Phong-shaded, metal-shaded (with and without edge lines), and metal-shaded with a cool-warm hue shift in Figure 10. The metal-shaded object is more obviously metal than the Phong-shaded image. The cool-warm hue metal-shaded object is not quite as convincing as the achromatic image, but it is more visually consistent with the cool-warm matte shaded model of Section 4.2, so it is useful when both metal and matte objects are shown together. We note that our banding algorithm is very similar to the technique Williams applied to a clear drinking glass using image processing [25].

4.4 Approximation to new model

Our model cannot be implemented directly in high-level graphics packages that use Phong shading. However, we can use the Phong lighting model as a basis for approximating our model. This is in the spirit of the non-linear approximation to global illumination used by Walter et al. [24]. In most graphics systems (e.g. OpenGL) we can use negative colors for the lights. We can approximate Equation 2 by two lights in directions $\hat{\mathbf{l}}$ and $-\hat{\mathbf{l}}$ with intensities $(k_{warm} - k_{cool})/2$ and $(k_{cool} - k_{warm})/2$ respectively, and an ambient term of $(k_{cool} + k_{warm})/2$. This assumes the object color is set to white. We turn off the Phong highlight because the negative blue light causes jarring artifacts. Highlights could be added on systems with accumulation buffers [11].

This approximation is shown compared to traditional Phong shading and the exact model in Figure 11. Like Walter et al., we need different light colors for each object. We could avoid these artifacts by using accumulation techniques which are available in many graphics libraries.

Edge lines for highly complex objects can be generated interactively using Markosian et al.'s technique [14]. This only works for polygonal objects, so higher-order geometric models must be tessellated to apply that technique. On high-end systems, image-processing techniques [19] could be made interactive. For metals on a conventional API, we cannot just use a light source. However, either environment maps or texture maps can be used to produce alternating light and dark stripes.

5 Future Work and Conclusion

The shading algorithm presented here is exploratory, and we expect many improvements are possible. The most interesting open ended question in automatic technical illustrations is how illustration rules may change or evolve when illustrating a scene instead of single objects, as well as the practical issues involved in viewing and interacting with 3D technical illustrations. It may also be possible to automate other application-specific illustration forms, such as medical illustration.

The model we have presented is tailored to imitate colored technical drawings. Once the global parameters of the model are set, the technique is automatic and can be used in place of traditional illumination models. The model can be approximated by interactive graphics techniques, and should be useful in any application where communicating shape and function is paramount.

Acknowledgments

Thanks to Bill Martin, David Johnson, Brian Smits and the members of the University of Utah Computer Graphics groups for help in the initial stages of the paper, to Richard Coffey for helping to get the paper in its final form, to Bill Thompson for getting us to reexamine our interpretation of why some illustration rules might apply, to Susan Ashurst for the sharing her wealth of artistic knowledge, to Dan Kersten for valuable pointers into the perception literature, and to Jason Herschaft for the dinosaur claw model. This work was originally inspired by the talk by Jane Hurd in the SIGGRAPH 97 panel on medical visualization. This work was supported in part by DARPA (F33615-96-C-5621) and the NSF Science and Technology Center for Computer Graphics and Scientific Visualization (ASC-89-20219). All opinions, findings, conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agencies.

References

[1] Irving Biederman and Ginny Ju. Surface versus Edge-Based Determinants of Visual Recognition. *Cognitive Psychology*, 20:38–64, 1988.

[2] Faber Birren. *Color Perception in Art.* Van Nostrand Reinhold Company, 1976.

[3] Wendy L. Braje, Bosco S. Tjan, and Gordon E. Legge. Human Efficiency for Recognizing and Detecting Low-pass Filtered Objects. *Vision Research*, 35(21):2955–2966, 1995.

[4] Tom Browning. *Timeless Techniques for Better Oil Paintings.* North Light Books, 1994.

[5] Chris Christou, Jan J. Koenderink, and Andrea J. van Doorn. Surface Gradients, Contours and the Perception of Surface Attitude in Images of Complex Scenes. *Perception*, 25:701–713, 1996.

[6] Cassidy J. Curtis, Sean E. Anderson, Kurt W. Fleischer, and David H. Salesin. Computer-Generated Watercolor. In *SIGGRAPH 97 Conference Proceedings*, August 1997.

[7] Debra Dooley and Michael F. Cohen. Automatic Illustration of 3D Geometric Models: Surfaces. *IEEE Computer Graphics and Applications*, 13(2):307–314, 1990.

[8] Gershon Elber and Elaine Cohen. Hidden Curve Removal for Free-Form Surfaces. In *SIGGRAPH 90 Conference Proceedings*, August 1990.

[9] E. Bruce Goldstein. *Sensation and Perception.* Wadsworth Publishing Co., Belmont, California, 1980.

[10] Paul Haeberli. Paint By Numbers: Abstract Image Representation. In *SIGGRAPH 90 Conference Proceedings*, August 1990.

[11] Paul Haeberli. The Accumulation Buffer: Hardware Support for High-Quality Rendering. *SIGGRAPH 90 Conference Proceedings*, 24(3), August 1990.

[12] Patricia Lambert. *Controlling Color: A Practical Introduction for Designers and Artists*, volume 1. Everbest Printing Company Ltd., 1991.

[13] Peter Litwinowicz. Processing Images and Video for an Impressionistic Effect. In *SIGGRAPH 97 Conference Proceedings*, August 1997.

[14] L. Markosian, M. Kowalski, S. Trychin, and J. Hughes. Real-Time Non-Photorealistic Rendering. In *SIGGRAPH 97 Conference Proceedings*, August 1997.

[15] Judy Martin. *Technical Illustration: Materials, Methods, and Techniques*, volume 1. Macdonald and Co Publishers, 1989.

[16] Barbara J. Meier. Painterly Rendering for Animation. In *SIGGRAPH 96 Conference Proceedings*, August 1996.

[17] Bui-Tuong Phong. Illumination for Computer Generated Images. *Communications of the ACM*, 18(6):311–317, June 1975.

[18] Tom Ruppel, editor. *The Way Science Works*, volume 1. MacMillan, 1995.

[19] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3D Shapes. In *SIGGRAPH 90 Conference Proceedings*, August 1990.

[20] Mike Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable Textures for Image-Based Pen-and-Ink Illustration. In *SIGGRAPH 97 Conference Proceedings*, August 1997.

[21] Doree Duncan Seligmann and Steven Feiner. Automated Generation of Intent-Based 3D Illustrations. In *SIGGRAPH 91 Conference Proceedings*, July 1991.

[22] Bosco S. Tjan, Wendy L. Braje, Gordon E. Legge, and Daniel Kersten. Human Efficiency for Recognizing 3-D Objects in Luminance Noise. *Vision Research*, 35(21):3053–3069, 1995.

[23] Edward Tufte. *Visual Explanations.* Graphics Press, 1997.

[24] Bruce Walter, Gun Alppay, Eric P. F. Lafortune, Sebastian Fernandez, and Donald P. Greenberg. Fitting Virtual Lights for Non-Diffuse Walkthroughs. In *SIGGRAPH 97 Conference Proceedings*, pages 45–48, August 1997.

[25] Lance Williams. Shading in Two Dimensions. *Graphics Interface '91*, pages 143–151, 1991.

[26] Georges Winkenbach and David H. Salesin. Computer Generated Pen-and-Ink Illustration. In *SIGGRAPH 94 Conference Proceedings*, August 1994.

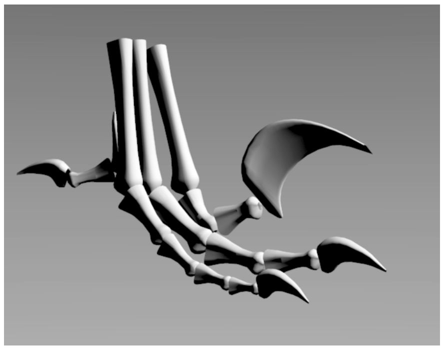


Figure 3: Diffuse shaded image using Equation 1 with $k_d = 1$ and $k_a = 0$. Black shaded regions hide details, especially in the small claws; edge lines could not be seen if added. Highlights and fine details are lost in the white shaded regions.

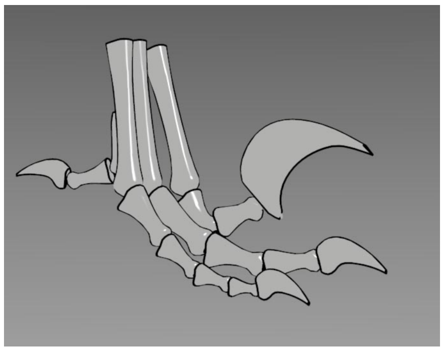


Figure 4: Image with only highlights and edges. The edge lines provide divisions between object pieces and the highlights convey the direction of the light. Some shape information is lost, especially in the regions of high curvature of the object pieces. However, these highlights and edges could not be added to Figure 3 because the highlights would be invisible in the light regions and the silhouettes would be invisible in the dark regions.

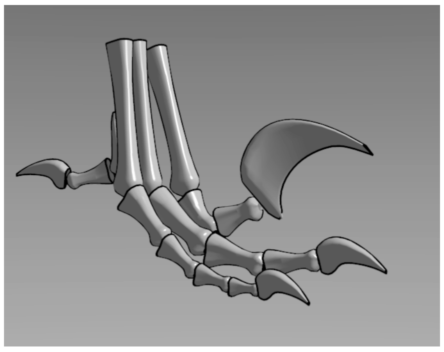


Figure 5: Phong shaded image with edge lines and $k_d = 0.5$ and $k_a = 0.1$. Like Figure 3, details are lost in the dark grey regions, especially in the small claws, where they are colored the constant shade of $k_d k_a$ regardless of surface orientation. However, edge lines and highlights provide shape information that was gained in Figure 4, but couldn't be added to Figure 3.

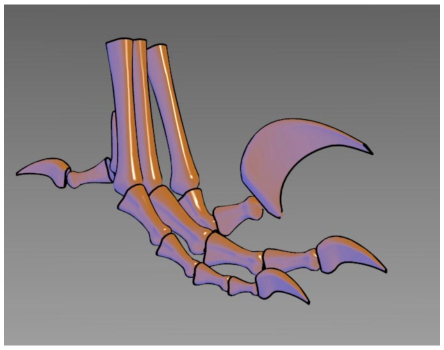


Figure 6: Approximately constant luminance tone rendering. Edge lines and highlights are clearly noticeable. Unlike Figures 3 and 5 some details in shaded regions, like the small claws, are visible. The lack of luminance shift makes these changes subtle.

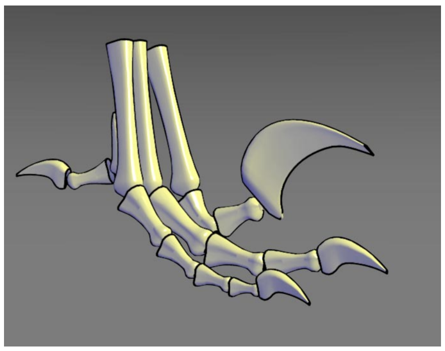


Figure 7: Luminance/hue tone rendering. This image combines the luminance shift of Figure 3 and the hue shift of Figure 6. Edge lines, highlights, fine details in the dark shaded regions such as the small claws, as well as details in the high luminance regions are all visible. In addition, shape details are apparent unlike Figure 4 where the object appears flat. In this figure, the variables of Equation 2 and Equation 3 are: $b = 0.4$, $y = 0.4$, $\alpha = 0.2$, $\beta = 0.6$.

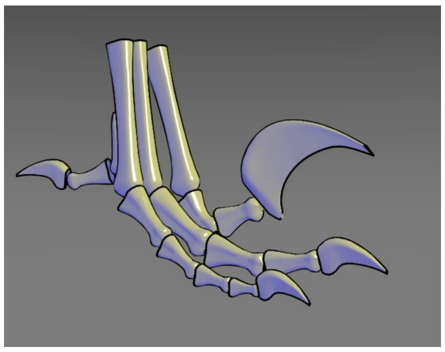


Figure 8: Luminance/hue tone rendering, similar to Figure 7 except $b = 0.55$, $y = 0.3$, $\alpha = 0.25$, $\beta = 0.5$. The different values of b and y determine the strength of the overall temperature shift, where as α and β determine the prominence of the object color, and the strength of the luminance shift.

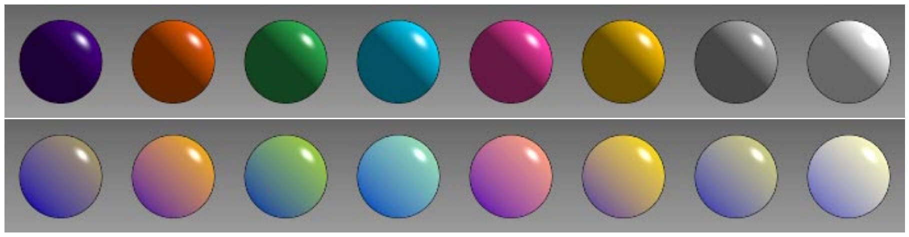


Figure 9: Top: Colored Phong-shaded spheres with edge lines and highlights. Bottom: Colored spheres shaded with hue and luminance shift, including edge lines and highlights. Note: In the first Phong shaded sphere (violet), the edge lines disappear, but are visible in the corresponding hue and luminance shaded violet sphere. In the last Phong shaded sphere (white), the highlight vanishes, but is noticed in the corresponding hue and luminance shaded white sphere below it. The spheres in the second row also retain their "color name".

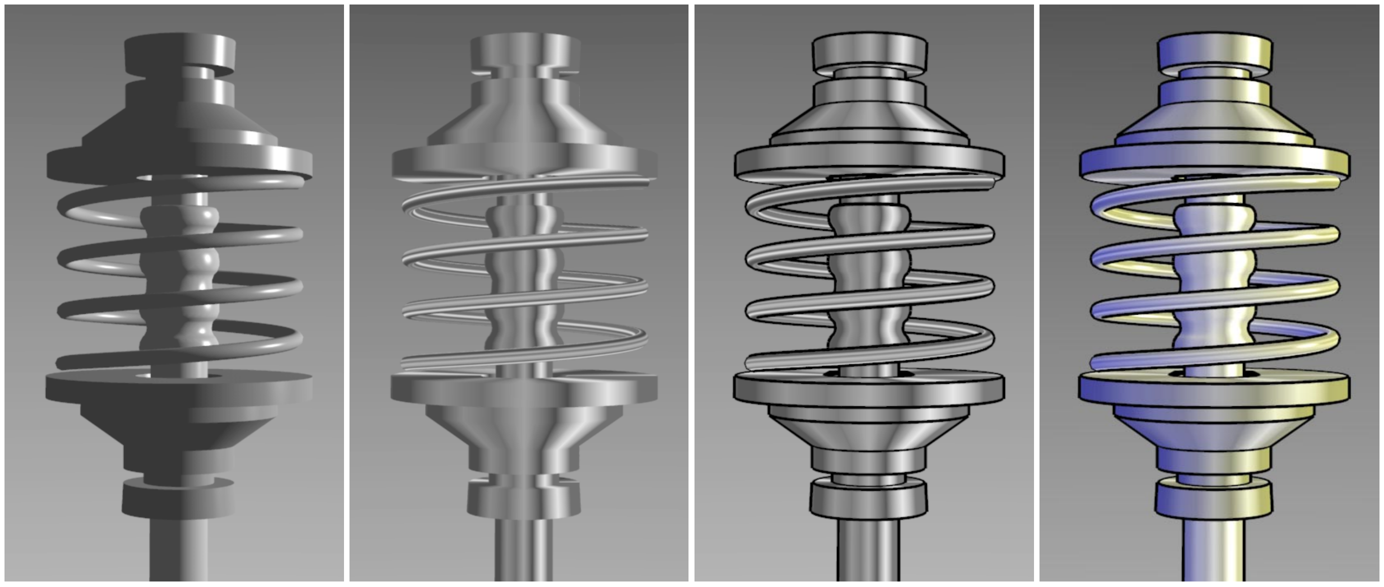


Figure 10: Left to Right: a) Phong shaded object. b) New metal-shaded object without edge lines. c) New metal-shaded object with edge lines. d) New metal-shaded object with a cool-to-warm shift.

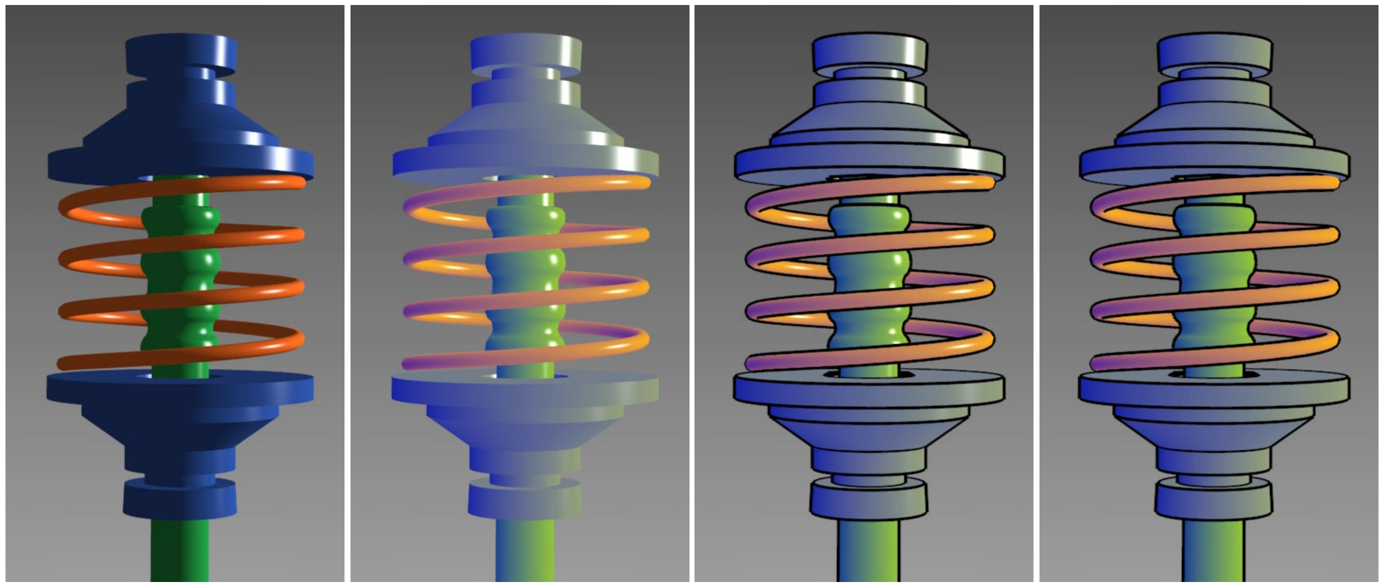


Figure 11: Left to Right: a) Phong model for colored object. b) New shading model with highlights, cool-to-warm hue shift, and without edge lines. c) New model using edge lines, highlights, and cool-to-warm hue shift. d) Approximation using conventional Phong shading, two colored lights, and edge lines.

Interactive Visualization Of 3D-Vector Fields Using Illuminated Stream Lines

Malte Zöckler, Detlev Stalling, Hans-Christian Hege

Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)[1]

Abstract

A new technique for interactive vector field visualization using large numbers of properly illuminated stream lines is presented. Taking into account ambient, diffuse, and specular reflection terms as well as transparency, we employ a realistic shading model which significantly increases quality and realism of the resulting images. While many graphics workstations offer hardware support for illuminating surface primitives, usually no means for an accurate shading of line primitives are provided. However, we show that proper illumination of lines can be implemented by exploiting the texture mapping capabilities of modern graphics hardware. In this way high rendering performance with interactive frame rates can be achieved. We apply the technique to render large numbers of integral curves in a vector field. The impression of the resulting images can be further improved by making the curves partially transparent. We also describe methods for controlling the distribution of stream lines in space. These methods enable us to use illuminated stream lines within an interactive visualization environment.

1 Introduction

The visual representation of vector fields is subject of ongoing research in scientific visualization. A number of sophisticated methods has been proposed to tackle this problem, ranging from particle tracing [7, 15, 11] over icon based methods [8, 13] to texture based approaches [3, 2, 4, 14, 9]. A straightforward, popular and still very powerful method is the concept of depicting stream lines. However, when using stream lines for visualization the user is confronted with a number of problems. First, on a common graphics workstation stream lines either have to be displayed using flat-shaded line segments, impairing the spatial impression of the image, or they have to be represented by polygonal tubes, strongly limiting the number of stream lines that can be displayed in a scene. Second, it is usually not quite obvious how to distribute stream lines in space in order to get expressive pictures without missing important details of the field.

In this paper we present ideas that can help to overcome both problems. To achieve a fast and accurate illumination of line segments we exploit the texture mapping capabilities of modern graphics hardware. We apply this new shading technique to render large numbers of stream lines distributed throughout a vector field. Taking into account light reflection on stream lines is of great significance for scientific visualization because it very much increases the spatial impression of the resulting images. Image quality can be further improved by making parts of a stream line semi-transparent. This allows us to get a better understanding of the inner structure of a field. It also makes it possible to distinguish between forward and backward direction. To facilitate the placement of a large number of stream lines we employ statistical methods. Given some scalar quantity that loosely describes the degree of interest in the vector field at some location, stream lines are placed automatically such that the relative degree of interest is matched qualitatively.

It is a well-known fact that quality and realism of computer generated images depend to a high degree on the accurate modeling of light interacting with the objects in a scene. Shading effects are perhaps the most important cue for spatial perception. Consequently much research has been performed to develop realistic illumination and reflection models in computer graphics. A widely used compromise between computational complexity and resulting realism is Phong's reflection model [12] which assumes point light sources and approximates the most important reflection terms by simple expressions. Traditionally the model is applied to surface elements. Today many graphics workstations offer hardware support for this kind of illumination. However, the model can also be generalized to line primitives, and in this paper we will make direct use of such a generalization.

In scientific visualization the goal is not to render natural scenes in a photo-realistic way, but to generate images which provide maximal insight into numerical or experimental data. Nevertheless, shading effects are at least as important for the spatial interpretation of artificial images as in traditional computer graphics. Shading provides the observer with a minimum of realism in a world of cutting planes, isosurfaces, and symbols. Unfortunately there are a number of visualization techniques which aren't based on surface primitives, and which therefore can't make use of the hardware shading capabilities of current graphics workstations. As an example consider the various volume rendering techniques. While interactive frame rates can be achieved

[1] Takustr. 7, D-14195 Berlin, Germany
E-mail: `{zoeckler,stalling,hege}@zib-berlin.de`

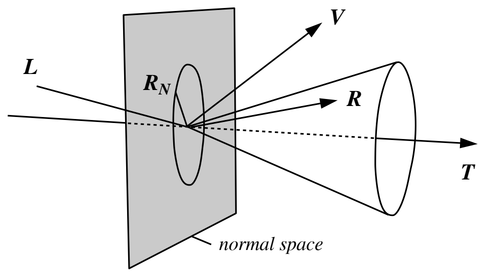


Figure 1: For line primitives there are infinitely many possible reflection vectors $\boldsymbol{R}$ lying on a cone around $\boldsymbol{T}$. For the actual lighting calculation we choose the one contained in the $\boldsymbol{L}$-$\boldsymbol{T}$-plane.

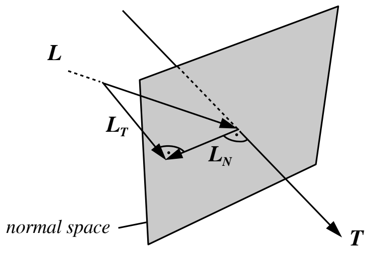


Figure 2: The light vector $\boldsymbol{L}$ can be decomposed into two orthogonal components $\boldsymbol{L}_T$ and $\boldsymbol{L}_N$ corresponding to the projection on the line's tangent and normal space, respectively.

for simple emission-absorption models by exploiting graphics hardware, in general this isn't yet possible if some sort of gradient dependent shading is included. Although rendering of line primitives is not as complex as volume rendering, the situation is similar. Traditionally, either flat shading has to be used or significant parts of the illumination calculation have to be computed without support by dedicated hardware.

After discussing illumination of line primitives in more detail, in section 3 we show how it can be implemented using texture mapping techniques. In section 4 we describe how to distribute stream lines in space in order to enhance interesting features within a vector field. In the final sections we present results and conclusions.

2 Illumination of Lines

Surfaces can be characterized locally by a distinct outward normal vector $\boldsymbol{N}$. This normal vector plays an important role when describing the interaction of light with surface elements. In the following we will shortly review the popular reflection model of Phong. Let $\boldsymbol{L}$ denote the light direction, $\boldsymbol{V}$ the viewing direction and $\boldsymbol{R}$ the unit reflection vector (the vector in the $\boldsymbol{L}$-$\boldsymbol{N}$-plane with the same angle to the surface normal as the incident light). Then light intensity at a particular surface point is given by

$$\begin{aligned} I &= I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}} \\ &= k_a + k_d\, \boldsymbol{L}\cdot\boldsymbol{N} + k_s\, (\boldsymbol{V}\cdot\boldsymbol{R})^n \end{aligned} \tag{1}$$

The first term, a global one, represents the ambient light intensity due to multiple reflections in the environment. The second term describes diffuse reflection due to Lambert's law. Diffuse light intensity does not depend on the viewing vector, i.e. diffuse reflecting objects look equally bright from all directions. The last term in Eq. (1) describes specular reflections on a surface. Specular reflections or highlights are centered around the reflection vector $\boldsymbol{R}$. The width of the highlights is controlled by the exponent n, also called shininess.

Let us now consider line primitives. In this case we can no longer define unique normal and reflection vectors. Instead there are two-dimensional manifolds containing infinitely many possible normal and reflection vectors. Mathematically lines in $\mathbb{R}^3$ are said to have codimension 2. Fortunately common surface reflection models can be generalized to higher codimensions in a straightforward way. These generalizations have been discussed in detail by Banks [1]. For lines in $\mathbb{R}^3$ the results are quite obvious. From all possible normal vectors we simply have to select the one which is coplanar to the light vector $\boldsymbol{L}$ and the tangent vector $\boldsymbol{T}$. Taking this particular normal vector we compute the diffuse reflection term as for surfaces using Eq. (1). Likewise, from all possible reflection vectors we choose the one coplanar to $\boldsymbol{L}$ and $\boldsymbol{T}$. Again, taking this particular reflection vector we use Eq. (1) to compute the specular reflection term. The relevant vectors for line illumination are illustrated in Fig. 1.

Instead of relying onto a specially selected normal vector we would rather like to express diffuse light intensity for line segments solely in terms of $\boldsymbol{L}$ and $\boldsymbol{T}$. Therefore we first project the light vector into the line's normal and tangent spaces, yielding an orthogonal decomposition $\boldsymbol{L} = \boldsymbol{L}_N + \boldsymbol{L}_T$. As illustrated in Fig. 2, by applying Pythagoras's theorem we obtain

$$\boldsymbol{L}\cdot\boldsymbol{N} = |\boldsymbol{L}_N| = \sqrt{1-|\boldsymbol{L}_T|^2} = \sqrt{1-(\boldsymbol{L}\cdot\boldsymbol{T})^2}. \tag{2}$$

Using similar arguments we can express the inner product $\boldsymbol{V}\cdot\boldsymbol{R}$ responsible for specular reflection solely in terms of $\boldsymbol{L}$, $\boldsymbol{V}$, and $\boldsymbol{T}$, i.e. without refering to $\boldsymbol{N}$. First, observe that $\boldsymbol{R}_N = -\boldsymbol{L}_N$ and $\boldsymbol{R}_T = \boldsymbol{L}_T$. We therefore have

$$\begin{aligned} \boldsymbol{V}\cdot\boldsymbol{R} &= \boldsymbol{V}\cdot(\boldsymbol{L}_T - \boldsymbol{L}_N) \\ &= \boldsymbol{V}\cdot((\boldsymbol{L}\cdot\boldsymbol{T})\boldsymbol{T} - (\boldsymbol{L}\cdot\boldsymbol{N})\boldsymbol{N}) \\ &= (\boldsymbol{L}\cdot\boldsymbol{T})(\boldsymbol{V}\cdot\boldsymbol{T}) - (\boldsymbol{L}\cdot\boldsymbol{N})(\boldsymbol{V}\cdot\boldsymbol{N}) \\ &= (\boldsymbol{L}\cdot\boldsymbol{T})(\boldsymbol{V}\cdot\boldsymbol{T}) - \\ &\quad \sqrt{1-(\boldsymbol{L}\cdot\boldsymbol{T})^2}\sqrt{1-(\boldsymbol{V}\cdot\boldsymbol{T})^2}. \end{aligned} \tag{3}$$

Here we have replaced $\boldsymbol{L}\cdot\boldsymbol{T}$ by Eq. (2). A similar expression has been used to rewrite $\boldsymbol{V}\cdot\boldsymbol{T}$.

3 Rendering

Despite the fact that the illumination equation looks the same for lines and surfaces, use of standard hardware shading techniques is impaired because for each new view or light direction a suitable normal vector has to be computed without utilizing graphics hardware. In the following we show how Eqs. (2) and (3) can be effectively evaluated using texture mapping capabilities of modern graphics hardware, avoiding the need of explicit normal vector computation. The technique allows us to achieve high frame rates even when large numbers of line segments have to be rendered.

3.1 Texture Mapping

We assume to have a graphics API available similar to OpenGL. In this graphics library at each vertex a homogeneous vector of texture coordinates can be specified. Usually the first components of this vector are taken as indices into a one-, two-, or three-dimensional texture map. A texture map may contain colors and/or transparencies which can be used to modify in various ways the original color of a fragment in the graphics pipeline. In addition it is possible to change texture coordinates using a 4×4 texture transformation matrix. This texture transformation is the key feature which makes it possible to employ texture mapping hardware for shading calculations.

3.2 Diffuse Reflection

Looking at Eq. (2) we note that the diffuse light intensity of a line segment is a function of $\boldsymbol{L}\cdot\boldsymbol{T}$ only. Specifying a texture vector $\boldsymbol{t}_0$ equal to the line's tangent vector $\boldsymbol{T}$ at each vertex, this inner product can be computed in hardware using the following texture transformation matrix:

$$M = \frac{1}{2}\begin{pmatrix} \boldsymbol{L}_1 & 0 & 0 & 0 \\ \boldsymbol{L}_2 & 0 & 0 & 0 \\ \boldsymbol{L}_3 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix}$$

The first component of the transformed homogeneous texture vector $\boldsymbol{t} = \boldsymbol{t}_0 M$ then evaluates to

$$t_1 = \frac{1}{2}(\boldsymbol{L}\cdot\boldsymbol{T} + 1).$$

Note, that t_1 always lies in the range $0 \ldots 1$. Therefore this value can be used as an index into a one-dimensional texture map $P(t_1)$. The value of the texture map at location t_1 is chosen such that it resembles the diffuse light intensity corresponding to $\boldsymbol{L}\cdot\boldsymbol{T} = 2t_1 - 1$, namely

$$P(t_1) = I_{\text{diffuse}} = k_d\,\sqrt{1 - (2t_1 - 1)^2}. \qquad (4)$$

Using a texture mode which takes the color of a line fragment to be equal to its texture color $P(t_1)$ we obtain an image which accurately shows line segments diffusely illuminated by a single point light source. If the light direction changes we simply have to update the texture transformation matrix. Vertices and texture coordinates of the line segments remain constant. This means that we can make use of OpenGL display lists to further increase rendering speed. Display lists allow one to specify multiple vertex and texture definitions using a single graphics library call.

3.3 Specular Reflection

The specular reflection term does not only depend on $\boldsymbol{V}\cdot\boldsymbol{T}$ but also on $\boldsymbol{L}\cdot\boldsymbol{T}$, as can be seen from Eq. (3). To compute this additional inner product we initialize the second column of the texture transformation matrix with the current viewing direction:

$$M = \frac{1}{2}\begin{pmatrix} \boldsymbol{L}_1 & \boldsymbol{V}_1 & 0 & 0 \\ \boldsymbol{L}_2 & \boldsymbol{V}_2 & 0 & 0 \\ \boldsymbol{L}_3 & \boldsymbol{V}_3 & 0 & 0 \\ 1 & 1 & 0 & 2 \end{pmatrix}$$

While the first transformed texture component remains the same, for the second component we now get

$$t_2 = \frac{1}{2}(\boldsymbol{V}\cdot\boldsymbol{T} + 1).$$

In order to obtain the correct light intensity corresponding to $\boldsymbol{L}\cdot\boldsymbol{T} = 2t_1 - 1$ and $\boldsymbol{V}\cdot\boldsymbol{T} = 2t_2 - 1$ we can use a two-dimensional texture map $P(t_1, t_2)$. Adding a constant ambient term k_a as well as the diffuse contribution from Eq. (4) we can perform the whole shading calculation for a single light source in texture hardware. Fig. 3 shows an example of a resulting two-dimensional texture map. One can clearly identify the highlight appearing at different angle positions on top of a diffuse background. If no highlight was present color would not depend on the viewing direction $\boldsymbol{V}$, as stated by Lambert's law.

It is worthwhile to note that there is an important special case, which allows one to use a one-dimensional texture even when specular reflection is present. This is the case of a headlight, i.e. a point light source located at the same position as the camera. In this case light vector and viewing vector are identical. Equation (3) simplifies to

$$\boldsymbol{V}\cdot\boldsymbol{R} = 2(\boldsymbol{L}\cdot\boldsymbol{T})^2 - 1.$$

Headlights are quite useful because they always guarantee an adequate illumination of the scene, irrespectively of the actual viewing direction. The user has not to bother with a tedious setup of light conditions. In fact, all of the color plates in this paper were rendered using a headlight.

Of course it is also possible to use the third column of the texture transformation matrix to compute an additional inner product. This would require the use of a three-dimensional texture map. Three different inner products would allow the

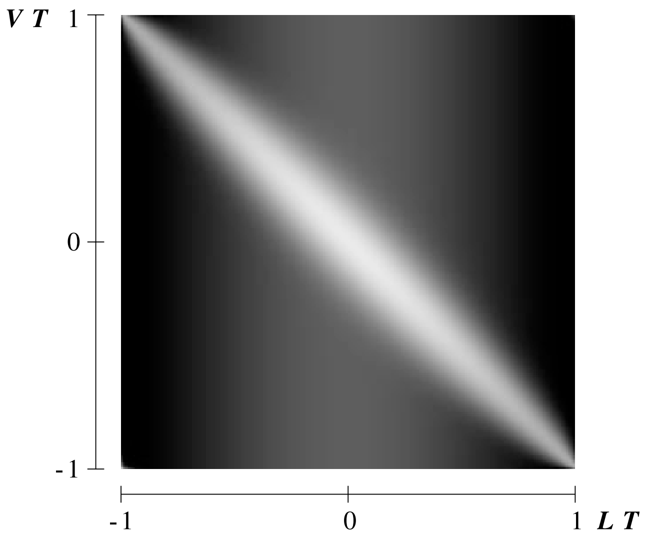


Figure 3: Two-dimensional texture map used to implement Phong's reflection model for line segments. Parameter values are $k_a = 0.1, k_d = 0.3, k_s = 0.6$, and $n = 40$.

illumination of lines by two point light sources located at arbitrary positions including specular reflection. Alternatively one might discard specular reflection and instead introduce a third purely diffuse illuminating light source.

3.4 Color Coding

Color coding is a common method in visualization. Applying color to individual field lines would enable us to depict some scalar quantity in addition to vector field structure. Such a quantity could be field magnitude or potential strength, or something more unrelated like pressure in a fluid flow. Ideally we would like to modify the curve's ambient and diffuse color components according to a given color lookup table. However, in our case color is directly taken from a texture map. Since we use the same texture map for all field lines it is not possible to set these components locally in a straight-forward way. Nevertheless, by using an alternative texture mapping mode it is possible to modulate, i.e. multiply, texture color with the object's base color. The latter can be defined for each vertex separately. This yields the desired effect with the restriction that also the specular highlight gets colored instead of remaining constant. Fig. 7 suggests that this is only a minor limitation. Despite being differently colored the highlight can be identified clearly throughout the whole image while still improving spatial perception. At the same time color accurately encodes an additional scalar variable.

3.5 Excess Brightness

Banks [1] pointed out that there is a general problem when illuminating objects with codimension > 1. The overall intensity of an image increases and becomes more uniform, thus disturbing spatial perception. In case of lines in $\mathbb{R}^3$ this can be understood by the following consideration: We know that the normal vector is not a constant one, but is given by the projection of the light vector into the line's normal space. Choosing such a vector means minimizing the angle between light vector and normal. Therefore in general the angle between these two vectors is smaller compared to the case of a fixed normal. This results in a more uniform brightness than we are used to perceive in real world. As suggested by Banks, we compensate the effect qualitatively by exponentiating the diffuse intensity term:

$$\hat{I}_{\text{diffuse}} = k_d\,(\boldsymbol{L}\cdot\boldsymbol{N})^p \tag{5}$$

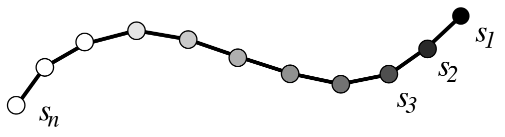
In [1] a value of $p = 4.8$ was proposed. For the images in this paper we have used a value of $p = 2$, which produced nicer results.

3.6 Transparency

Shading of line segments as described above provides important cues for the spatial impression of stream line images. However, image quality can be further improved by use of transparency. Let us imagine the image of a stream line is produced by a small particle traversing the vector field and leaving a veil of haze. Assuming that the haze disappears according to an exponential law, opacity or alpha value for a point s_n at the curve is given by

$$\alpha(s_n) = \alpha_0 q^{n-1}. \tag{6}$$

Here the factor q controls how much of the haze disappears per unit step. A resulting semi-transparent stream line is illustrated in the following figure:



Use of transparency has two advantages: First, stream lines near to the camera do not completely hide those being more far away. This allows the observer to gain deeper insight into the inner structure of the vector field. Second, the sign of vector field direction becomes visible in a static image. This is not the case when stream lines are rendered symetrically in forward and backward direction.

Drawing a transparent pixel of opacity α and color C causes the current color in the frame buffer to be updated according to

$$C_{\text{new}} = (1-\alpha)C_{\text{old}} + \alpha C. \tag{7}$$

In general if multiple transparent objects are present the final color depends on the ordering of the individual objects. Correct results are obtained using a back to front traversal. The situation is simplified if all objects are of equal color C.

In this case all traversal orders yield the same result. This has been exploited by Max, Crawfis, and Grant [10], who applied constant shaded line bundles for vector field visualization. However, for illuminated lines color isn't constant anymore. Therefore individual lines have to be rendered in a depth-sorted way.

In general it is impossible to achieve an exact depth ordering for extended curves in 3D, because mutual coverings may occur. Therefore we split each stream line into many small line segments, which are sorted and rendered individually. To avoid resorting line segments each time the view direction changes we use the following simplified algorithm: Three lists of pointers to stream line segments are created. The lists are sorted in order of increasing x-, y-, and z-coordinates, respectively. During rendering the list that most closely resembles the viewing direction is traversed, either from back to front or from front to back. Although this method is not exact, it produces excellent results which can not be distinguished from the exact images visually. Experiments have shown, that only about 1% of all pixels receive somewhat incorrect color values.

3.7 Stream Line Animation

Animated particles provide a very intuitive mean of visualization, especially when velocity fields are to be visualized. Following the idea of particles leaving a veil of haze, animation sequences can be obtained in the following way.

Stream lines are created at different times t_i with an initial length of 0. In each time step, all stream lines are extended by one point, while opacity of all the points already drawn is modified by the factor q (compare Eq. (7)). This gives the illusion of moving particles producing a slowly disappearing veil of haze, like comets. A periodic animation sequence can be created by assuring that the period T is long enough so that points on a stream line can disappear completely within this interval (i.e. $q^T \approx 0$). Then a stream line that has been created at time t_i can be restarted at the same location at time $t_i + T$, since it is no longer visible then. This results in a continuous animation loop of period T.

4 Distributing Stream Lines

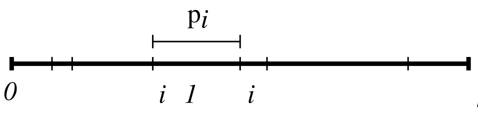
When using stream lines for vector field visualization a common problem is to select proper seed points for path tracking. The fast texture based shading technique described above allows us to render images containing thousands of stream lines at interactive rates. Working with a large number of stream lines has the advantage, that the positioning of an individual line becomes less important. Instead we can apply statistical methods to distribute seed points throughout the field. In particular we would like the distribution to resemble some sort of scalar quantity p, which loosely corresponds to the degree of interest the user wants to put in some region. For example a constant p would result in a homogenous distribution of seed points, while a value of p proportional to vector magnitude would have the effect that more seed points are placed in regions of large magintude.

4.1 Monte-Carlo Selection

To generate seed points with a density proportional to p, we subdivide to whole data volume into n uniform cells. For each cell we compute a value p_i describing the local degree of interest for that cell. The accumulated degree of interest is defined by

$$\omega_i = \sum_{j=1}^{i} p_j \tag{8}$$

We assume all cells being arranged in a sequence based on some arbitrary numbering. We choose cells randomly with a probability proportional to p_i. This is done by taking a random number r uniformly distributed in the range $0 \ldots \omega_n$. The first value $\omega_i > r$ determines which cell is taken.



Within a selected cell we place a new seed point at a random position. Because the values ω_i are monotonely increasing, the cell lookup procedure has a complexity of $O(\log(n))$ and therefore can be performed quite fast.

4.2 An Equalization Strategy

In general it is not a trivial task to find a good scalar quantity p. For example, choosing p equal to vector field magnitude may not have the desired effect when this quantity varies over multiple orders of magnitude. Instead of exactly being proportional to p, we would rather like to have a density distribution which resembles vector field magnitude qualitatively, but in general places seed points more homogenously. Such an effect can be obtained using a histogram equalization approach. This technique is well known from the image processing literature [6], but in our case may also be used to modify the degree of interest p in a suitable way. Let us define a sum histogram in the following way:

$$S(p) = \frac{\text{number of cells with } p_i < p}{\text{total number of cells}} \tag{9}$$

Based on the sum histogram we can assign each cell a new equalized degree of interest,

$$p'_i = S(p_i). \tag{10}$$

Of course other probability distributions can be used to emphasize special features of the field. We have implemented a symbolic interface which allows us to specify p_i as a function of vector field magnitude and other optional scalar fields. Within this interface analytic functions like logarithm or square root as well as threshold operators can be used to modify p. Together with a three-dimensional selection box, which may be positioned interactively to spatially confine the region of interest, it is possible to explore very quickly the overall characteristics as well as the details of a vector field.

4.3 Divergence Compensation

If the vector field has a divergence different from zero, the stream line density will not remain constant even if the seed points are distributed uniformily. In some areas stream lines will run together, resulting in an increased local density. In other areas they will expand, resulting in a decreased local density. In our case stream lines are computed with a fixed maximum length. Experience shows that a sufficiently uniform stream line density is obtained by placing seed points in the middle of a stream line segment, and integrating equally far in forward and backward direction. Of course, better results could be obtained by adaptively terminating existing lines or creating new ones based on local stream line density.

4.4 Streamline Computation

For numerical stream line integration we use a fourth-order Runge-Kutta method with error monitoring and adaptive step size control, as described in [14]. Use of an adaptive method allows us to control the error of the solution. Such methods are also necessary to detect singularities. At these points stream line integration has to be terminated. Singularities, i.e. sinks and sources, commonly occur for example in electrostatic fields. Examples are shown in Figs. 4-7.

5 Interaction

Due to its high rendering speed our method is dedicated to be used in an interactive visualization framework. To interactively define regions of interest we apply so-called draggers which are provided by the Open Inventor graphics toolkit. Two such box-type draggers are depicted in Fig. 4. Each dragger defines an rectangular or spherical volume in which stream lines are seeded.

We have also implemented methods to place seed points along curves. The curves may be created by intersecting an arbitrary geometry, e.g. an isosurface, and a user-defineable plane. This method is particularly suited to highlight possible symmetries within the data set.

Further refinements can be achieved by using scalar fields that define local seed point density or local degree of interest. Such fields can be obtained from numerical simulation. In addition we use a symbolic interface to define such fields in terms of vector field magnitude, cartesian coordinates and other available quantities.

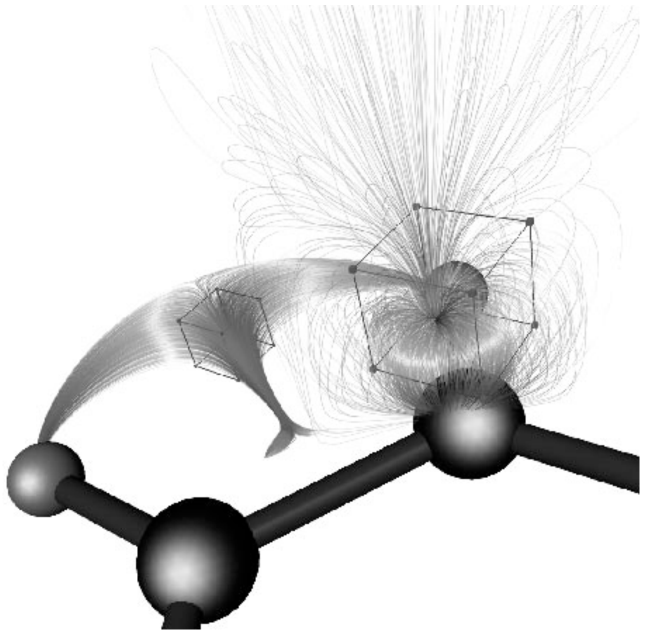


Figure 4: Interactive definition of seed volumes using Open Inventor draggers.

6 Results

The algorithms presented in this paper have been implemented in C++ by subclassing the Open Inventor toolkit. Using Inventor makes it easy to display shaded stream lines in combination with other geometries. The shading itself makes use of the OpenGL graphics library.

On a SGI Indigo^2 desktop workstation with Maximum Impact graphics and a 250 MHz R4400 CPU scenes containing 3000 stream lines each consisting of 120 transparent line segments can be rendered at a rate of 10 frames per second. These results can be improved by 10%-20% if OpenGL display lists are used. However, display lists cause the rendering to be delayed when the scene is drawn the first time. Therefore, in our implementation the user can choose whether to use display lists or not. Also, the integration of our algorithm into the Open Inventor rendering scheme may be further optimized.

We have applied our methods to visualize vector fields from various disciplines like computational fluid dynamics, quantum chemistry, and medicine. In most cases the default values for seed point distribution (eventually accompanied with the histogram equalization technique) provide a good first impression of the vector field. The fast rendering speed offers the possibility to interactively rotate and zoom the geometry. This is an important feature for an improved spatial perception.

Fig. 5 and 6 show the electrostatic field of a benzene molecule. The field is computed using the NAO-PC method (Natural Atomic Orbitals - Point Charge). This quantumclassical method aproximates atomic orbitals by a set of discrete fractional point charges. The location of some of these point charges can be clearly identified in the images.

Fig. 7 also depicts parts of an electrostatic field of a molecule. In addition stream lines have been color coded as described in section 3.4. In this example color depicts the electrostatic potential. The field lines connect several positive point charges (magenta) with a single negative charge (green-orange).

An example of a velocity field from a CFD application is shown in Fig. 8. The data represents a fluid flow over a backward facing step. The turbulent region emphasized by the visualization is characterized by a very complex field structure.

Finally Fig. 9 illustrates the power of accurate line shading: While only a poor three-dimensional impression is obtained from the middle and right images, the spatial structure of the field is clearly revealed in the left image.

7 Conclusion

The visual representation of 3D vector fields is one of the current challenges in scientific visualization. Of particular interest are methods that provide an overview of the global field structure and that also depict fine details.

In this paper we have presented a fast method for visualizing 3D vector fields based on the display of stream lines, i.e. integral curves of the field. The method gives a good impression ofthe field structure and enables us to resolve visually rather fine details, like small vortices. A texture mapping technique is used to accurately illuminate the stream lines. Light reflection on stream lines improves spatial perception and thereby facilitates the understanding of the inner structure of a field.

We have shown how high quality stream line images can be generated at interactive speed using hardware supported texture mapping. This offers new opportunities for interactive visualization. Using a simple Monte-Carlo method lines are placed automatically such that the relative degree of interest, defined by some scalar field, is matched qualitatively. Additional use of of a histogram equalization approach allows us to automatically place stream line segments more homogenously.

Some interesting topics of further research are improvement of the seed point selection strategies such that characteristic features of the field are detected and enhanced automatically or the application of the shading technique to time dependent vector fields. In the latter case particle paths or streak lines should be used in favour of stream lines.

References

[1] D.C. Banks, *Illumination in Diverse Codimensions*, Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24-29, 1994). In *Computer Graphics* Annual Conference Series, 1994, ACM SIGGRAPH, pp. 327-334.

[2] B. Cabral, L. Leedom, *Imaging vector fields using line integral convolution*, Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In *Computer Graphics* 27, 1993, ACM SIGGRAPH, pp. 263-272.

[3] Roger Crawfis, Nelson Max, *Textured Splats for 3D Scalar and Vector Field Visualization*, Proceedings of Visualization '93, Nielson and Bergeron, Eds., IEEE Computer Society Press, 1993, pp. 261-272.

[4] L.K. Forsell, *Visualizing Flow over Curvilinear Grid Surfaces unsing Line Integral Convolution*, Proceedings of Visualization '94, Bergeron and Kaufman, Eds., IEEE Computer Society Press, 1994, pp. 240-247.

[5] Allen Van Gelder, Jane Wilhelms, *Interactive Animated Visualization of Flow Fields*, Proceedings of ACM Workshop on Volume Visualization, 1992, pp. 47-54.

[6] R.C. Gonzales, P. Wintz, *Digital Image Processing*, Addison Wesley, Second Edition, 1987, pp. 146–152.

[7] Andrea J. S. Hin and Frits H. Post, *Visualization of turbulent flow with particles*. In *Visualization '93*, IEEE Computer Society Press, pp. 46-52.

[8] W.C. de Leeuw, J.J. van Wijk, *A probe for local flow field visualization*, Proceedings of Visualization '93, Nielson and Bergeron, Eds., IEEE Computer Society Press, 1993, 39-45.

[9] W.C. de Leeuw, J.J. van Wijk, *Enhanced Spot Noise for Vector Field Visualization*, Proceedings of Visualization '95, Nielson and Silver, Eds., IEEE Computer Society Press, 1995, pp. 233-239.

[10] N. Max, R. Crawfis, C. Grant, *Visualizing 3D Velocity Fields Near Contour Surfaces*, Proceedings of Visualization '94, Bergeron and Kaufman, Eds., IEEE Computer Society Press, 1994, pp. 248-255.

[11] Kwan-Liu Ma and Philip J. Smith, *Virtual smoke: An interactive 3d flow visualization technique*, In *Visualization '92*, IEEE Computer Society Press, pp. 46-52.

[12] Bui-T. Phong, *Illumination for Computer Generated Pictures*, Communications of the ACM, June 1975, pp. 311-317.

[13] F.J. Post, T. van Walsum, F.H. Post, *Iconic Techniques for Feature Visualization*, Nielson and Silver, Eds., Proceedings of Visualization '95, pp. 288-295.

[14] D. Stalling, H.C. Hege, *Fast and Resolution Independent Line Integral Convolution*, Proceedings of SIGGRAPH '95 (Los Angeles, California, August 6-11, 1995). In *Computer Graphics* Annual Conference Series, 1995, ACM SIGGRAPH, pp. 249-256.

[15] Jarke J. van Wijk, *Rendering surface-particles*, In *Visualization '92*, IEEE Computer Society Press, pp. 54-61.

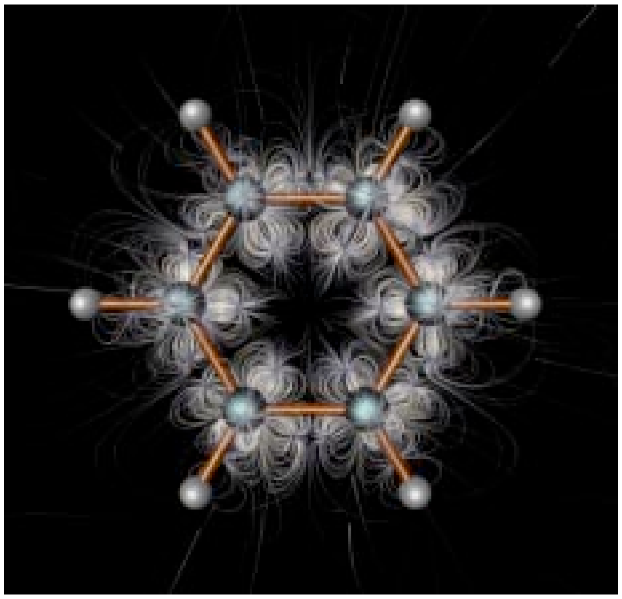


Fig.5: Electrostatic field of a benzene molecule.

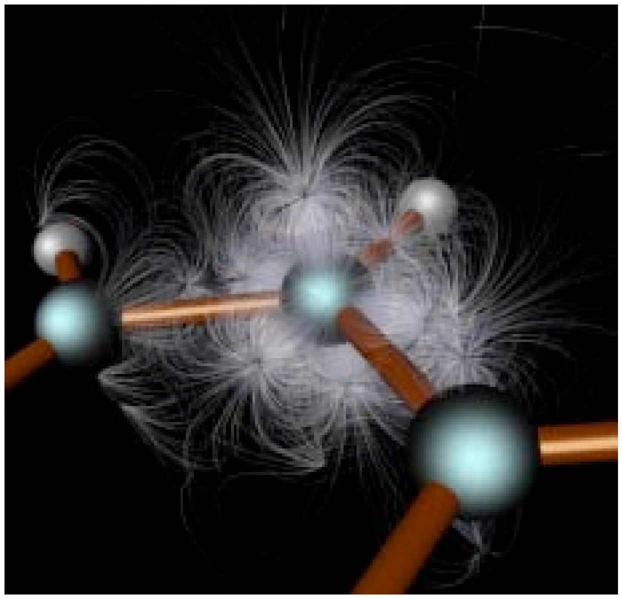


Fig.6: Detail view of the benzene.

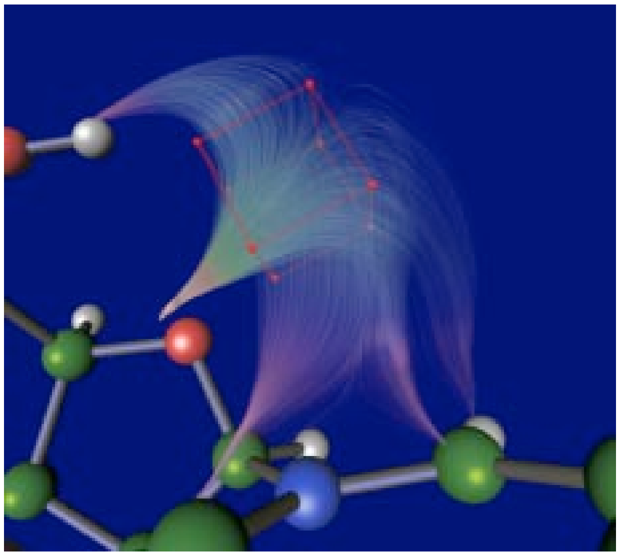


Fig.7: Example of illuminated colored stream lines.

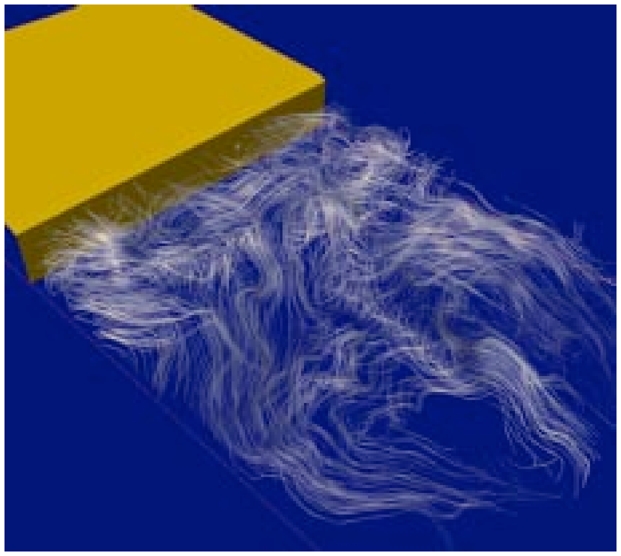


Fig.8: Velocity field from a CFD application.

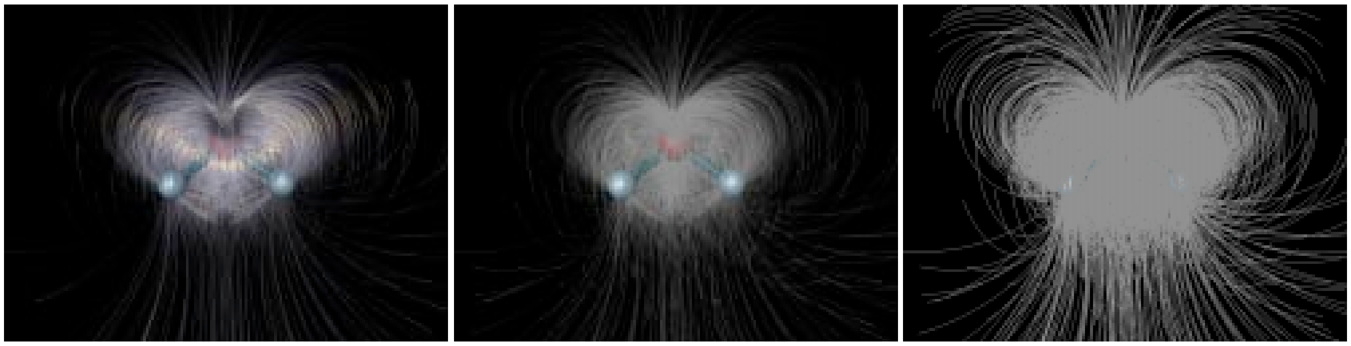


Fig.9: (a) Illuminated stream lines. (b) Flat shaded transparent stream lines. (c) Flat shaded opaque stream lines.