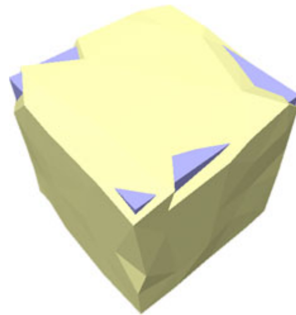
Computergraphik I

Susanne Krömker



Wintersemester 2009/10

Skript zur Vorlesung, Stand 1. Februar 2010

Inhaltsverzeichnis

1 Einführung **1**

1.1 Was ist Computergraphik? 1

1.2 Anwendungsbereiche 2

1.3 Kurzer geschichtlicher Abriss 3

1.4 Literaturhinweise 7

1.4.1 Fachzeitschriften 8

1.4.2 Buchtipps 8

1.5 Programmierhilfen 10

1.5.1 Kurzer Exkurs über `make` 10

1.5.2 Noch kürzerer Exkurs über Shellskripte 13

1.6 OpenGL 13

1.6.1 Einbinden von Bibliotheksfunktionen 14

1.6.2 Zusatzbibliotheken 15

1.6.3 Namenskonventionen 18

1.6.4 OpenGL ist eine *State machine* 19

1.7 Graphikpipeline 20

1.8 Übungsaufgaben 22

2 Koordinatensysteme 25

2.1 Welt-, Bild- und Gerätekoordinaten . . . 25

2.2 Transformationen . . . 27

2.3 Homogene Koordinaten . . . 29

2.3.1 OpenGL arbeitet mit homogenen Koordinaten ($w = 1$) . . . 30

2.4 Kameraanalogie . . . 30

2.4.1 Aufstellen der Kamera . . . 31

2.4.2 Arrangieren der Szene . . . 31

2.4.3 Linseneinstellung . . . 31

2.4.4 Belichten des Abzugs . . . 34

2.5 Matrix Stack in OpenGL . . . 34

2.5.1 Arbeiten mit dem Matrixstack . . . 36

2.6 Übungsaufgaben . . . 37

3 Zeichenalgorithmen 41

3.1 Zeichnen von Linien . . . 41

3.1.1 Midpoint Line Algorithmus . . . 41

3.1.2 Doppelschrittalgorithmus . . . 44

3.1.3 Unabhängigkeit der Laufrichtung . . . 45

3.1.4 Schnittpunkte mit Rechtecken . . . 46

3.1.5 Intensitätsschwankungen . . . 47

3.2 Zeichnen von Kreisen . . . 47

3.2.1 Symmetrieeigenschaften . . . 48

3.2.2 Midpoint Circle Algorithmus . . . 48

3.3 Antialiasing von Linien und Flächen . . . 51

3.3.1 Ungewichtete Flächenabtastung . . . 52
3.3.2 Gewichtete Flächenabtastung . . . 53
3.4 Das Zeichnen von Polygonen . . . 53
3.5 OpenGL Implementierung . . . 54
3.5.1 Gestrichelte Linien . . . 54
3.5.2 Antialiasing . . . 55
3.5.3 Linienbreite . . . 56
3.5.4 Zeichnen mit Vertices . . . 56
3.6 Übungsaufgaben . . . 58

4 Bufferkonzepte **61**
4.1 Double (und Stereo) Buffering . . . 61
4.1.1 Passiv Stereo für räumliches Sehen . . . 63
4.1.2 Aktiv Stereo für räumliche Interaktionen . . . 64
4.2 Depth- oder z-Buffer . . . 65
4.2.1 Der Depth Buffer Algorithmus . . . 67
4.2.2 Das Zeichnen von Polygonen aus Dreiecken . . . 69
4.2.3 Hidden Surface . . . 70
4.2.4 Hidden Line . . . 71
4.3 Stencil Buffer . . . 73
4.4 Accumulation Buffer . . . 74
4.5 Abfolge der Testoperationen . . . 75
4.6 Übungsaufgaben . . . 75

5 Farbe und Licht **79**

5.1 Achromatisches Licht 79
5.1.1 Intensitäten 80
5.1.2 Halbtonnäherung 81
5.2 Chromatisches Licht und Farbmodelle 82
5.2.1 Wahrnehmungsmodelle 82
5.2.2 Darstellungsmodelle 85
5.2.3 Übertragungsmodelle 88
5.3 Farbinterpolation und Schattierung 91
5.3.1 Gouraud Shading 91
5.3.2 Phong Shading 93
5.4 Phong Modell, ein einfaches Reflexionsmodell 95
5.4.1 Umgebungslicht 96
5.4.2 Diffuses Streulicht 97
5.4.3 Spiegelnde Reflexion 97
5.4.4 Blinn-Beleuchtungsmodell, eine Vereinfachung des Phong Modells 100
5.4.5 Abschwächung des Lichts durch Abstände 102
5.4.6 Grenzen der einfachen Reflexionsmodelle 103
5.5 Lichtquellen 104
5.5.1 Headlight 104
5.5.2 Punktförmige Lichtquellen 105
5.5.3 Gerichtete Lichtquellen 105
5.6 Zusammenspiel von Licht und Material in OpenGL 105
5.7 Übungsaufgaben 107

6 Texturen **109**

6.1 Eindimensionale Texturen . . . 111
6.2 Zweidimensionale Texturen . . . 112
6.2.1 UV-Mapping . . . 113
6.2.2 Perspektivkorrektur . . . 113
6.2.3 Entfaltung des Polygonnetzes . . . 114
6.2.4 Zweiteiliges Abbilden . . . 115
6.3 Dreidimensionale Texturen . . . 116
6.4 Bump Mapping . . . 117
6.4.1 Displacement Mapping . . . 118
6.5 Interpolationen und Skalierungen . . . 119
6.5.1 Mipmapping . . . 120
6.5.2 Ripmapping . . . 122
6.6 OpenGL Implementierung . . . 123
6.7 Übungsaufgaben . . . 124

7 Raytracing **125**

7.1 Raytracing arbeitet im Objektraum . . . 128
7.2 Historisches aus der Optik . . . 129
7.2.1 Farbaufspaltung . . . 129
7.3 Implementierung: Binärer Baum und Schachtelungstiefe . . . 131
7.3.1 Rekursiver Algorithmus . . . 132
7.3.2 Fortschritte des Verfahrens . . . 134
7.4 Schnittpunktbestimmung . . . 135
7.4.1 Schnittpunkte mit Kugeln . . . 135
7.4.2 Schnittpunkte mit Polygonen . . . 136

7.5 Beschleunigung des Algorithmus 137
7.5.1 Z-Sort 137
7.5.2 Begrenzende Volumen 137
7.5.3 Raumaufteilung - Fast Voxel Transversal Algorithm 138
7.5.4 Granulierung 138
7.5.5 Vergleich der Performance 140
7.6 Übungsaufgaben 141

A DIN A-Formate und 150 dpi-Umrechnung 143

Literaturverzeichnis 145

Kapitel 1

Einführung

1.1 Was ist Computergraphik?

Mit der Entwicklung der modernen Computer ist untrennbar die graphische Ausgabe am Bildschirm verbunden. Kaum jemand erinnert sich noch an die bildschirmlose Zeit oder an die reine Textausgabe auf dem Schirm. Da das menschliche Begreifen so sehr auf seinen visuellen Sinn ausgerichtet ist, wird Computergraphik in vielen Wissenschaften erfolgreich eingesetzt. Abstrakte Modellvorstellungen, die früher nur mühsam vermittelt werden konnten und in den Köpfen verbleiben mussten, stellen heute ihre Tauglichkeit am Bildschirm unter Beweis. Mit Captain Janeway gesprochen: ”... auf den Schirm!”

Der große Themenbereich dessen, was mit Computern graphisch behandelt wird, unterteilt sich in die drei Bereiche

- **Generative Computergraphik** = *Computer Graphics*
  Visualisierung von Simulationsrechnungen, Animationen (Transformationen und dynamische Entwicklungen), atmosphärische Effekte, *Virtual Reality (VR)*
- **Bildverarbeitung** = *Image Processing*
  2D Bildrestauration, Filterverfahren, Vervollständigung fehlender Bildteile, 3D Bearbeitung von Bildsequenzen (Video, Auswertung zeitabhängiger Messungen), Aufbereitung von Volumendaten, Bildsegmentierung, Auffinden von Hyperflächen
- **Bildanalyse** = *Picture Analysis*
  Bild- und Mustererkennung, *Augmented Reality*, Anwendungen im Prüfwesen, Geologie, Kartogaphie, Medizin

Die Vorlesung wird sich hauptsächlich mit der generativen Computergraphik beschäftigen, also dem

Erzeugen *künstlicher Bildwelten* aufgrund von *errechneten* oder *gemessenen* Daten und ihrer Manipulation am Rechner.

1.2 Anwendungsbereiche

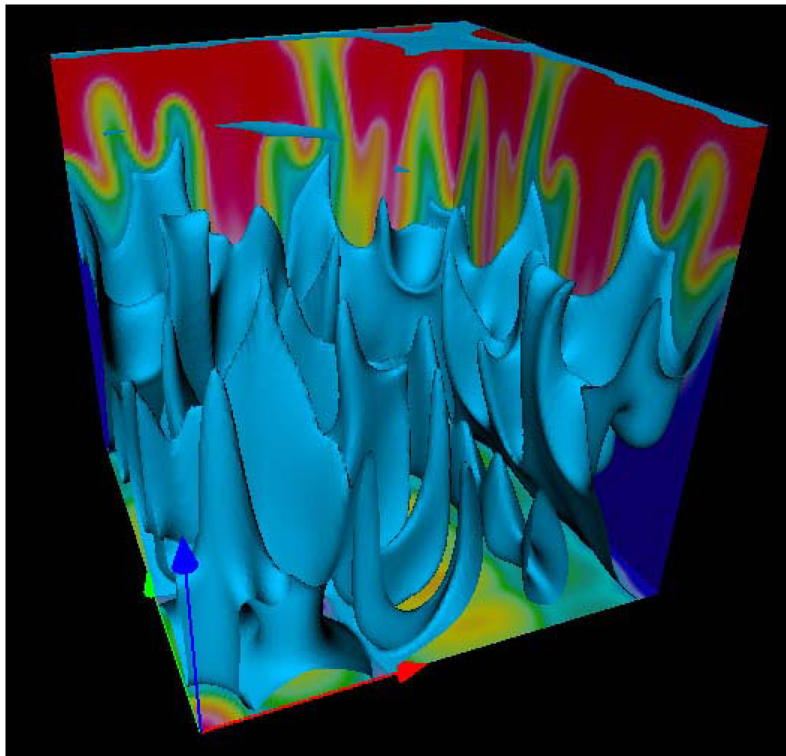


Abbildung 1.1. Zweiphasenströmung mit Isofläche, Peter Bastian, IWR, Universität Heidelberg

Die unterschiedlichen Anwendungen reichen dabei von interaktiven Bedienhilfen für Computerprogramme oder computergesteuerte Geräte und Großanlagen bis zu eigenständigen Visualisierungswerkzeugen. Meist werden die Bilder mit direktem Rendering in Echtzeit umgesetzt.

- *GUI* = Graphisches User Interface, wird heute von jedem Programm erwartet: Intuitiver und durch schnelles Feedback einfach erlernbarer Umgang mit der Software
- *Interaktive Darstellung* von Daten der Wirtschaft, Naturwissenschaft, Technik, hier insbesondere schnelle Aufbereitung und parameterabhängige Neuberechnung, Datenreduktion (insbesondere für schnelle Darstellung in VR)
- *Datenauswertung* in der Medizin, hier besonders Diagnostik und Operationsplanung (Minimalinvasive Chirurgie)

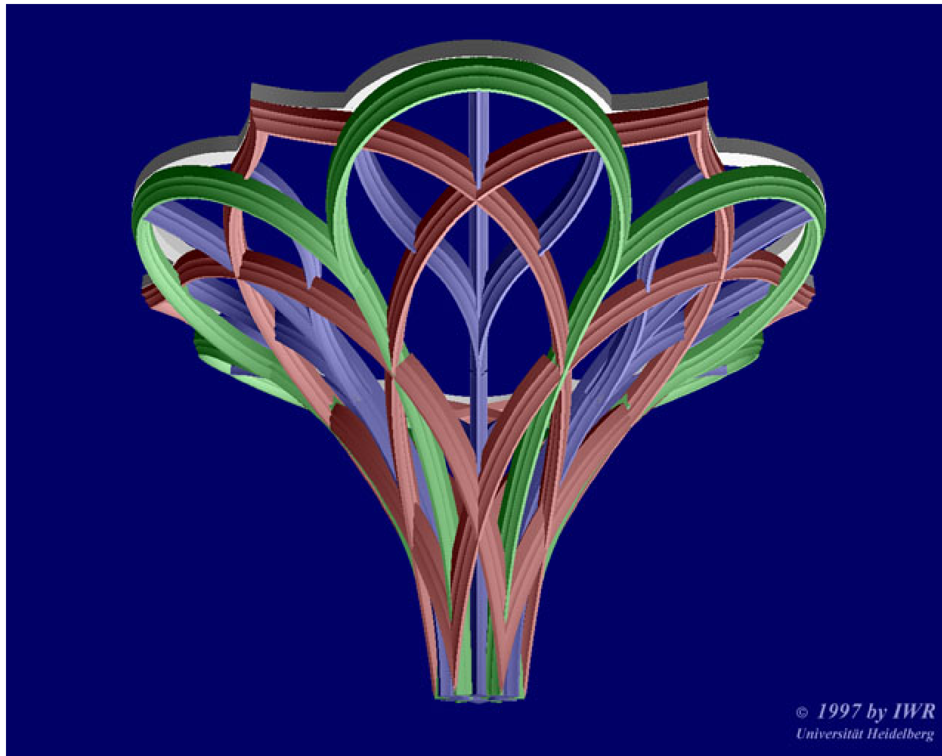


Abbildung 1.2. Mittels Raytracing dargestelltes spätgotisches Ziergewölbe, DFG Projekt, Werner Müller, Norbert Quien, Kurt Sätzler, IWR, Universität Heidelberg

- *Multimediale* Lernprogramme (Flugsimulatoren, Katastrophentraining mit VR z.B. Rauch im Cockpit, audiovisuelles Erlernen von Fremdsprachen) und Spielprogramme, Unterhaltungsindustrie (wesentlich an der Entwicklung schneller Graphikprozessoren beteiligt)

1.3 Kurzer geschichtlicher Abriss

bis 1950 Graphikausgabe: Plotterzeichnungen

1950 Am *Massachusetts Institute of Technology MIT* wird erstmals der Einsatz von Kathodenstrahlröhren *(Cathode Ray Tube CRT)* zur Ausgabe benutzt, fünf Jahre später militärischer Einsatz in der Luftabwehr der USA

1963 Beginn der modernen Computergraphik, Ivan Sutherland, Dissertation: *Sketches and Systems*, MIT. Sein Programm *Sketchpad* stellte in Zeiten der späten Lochkartenrechner einen ersten Schritt in Richtung graphische Schnittstelle dar, nutzte bereits einen frühen Röhrenbildschirm. Die Anwendung lief auf einem massiv umgebauten Lincoln TX2-Rechner, der nach Ende der Arbeiten zurückgebaut wurde. 1968-74 Professur an der University of Utah, 1974-78 am CalTech

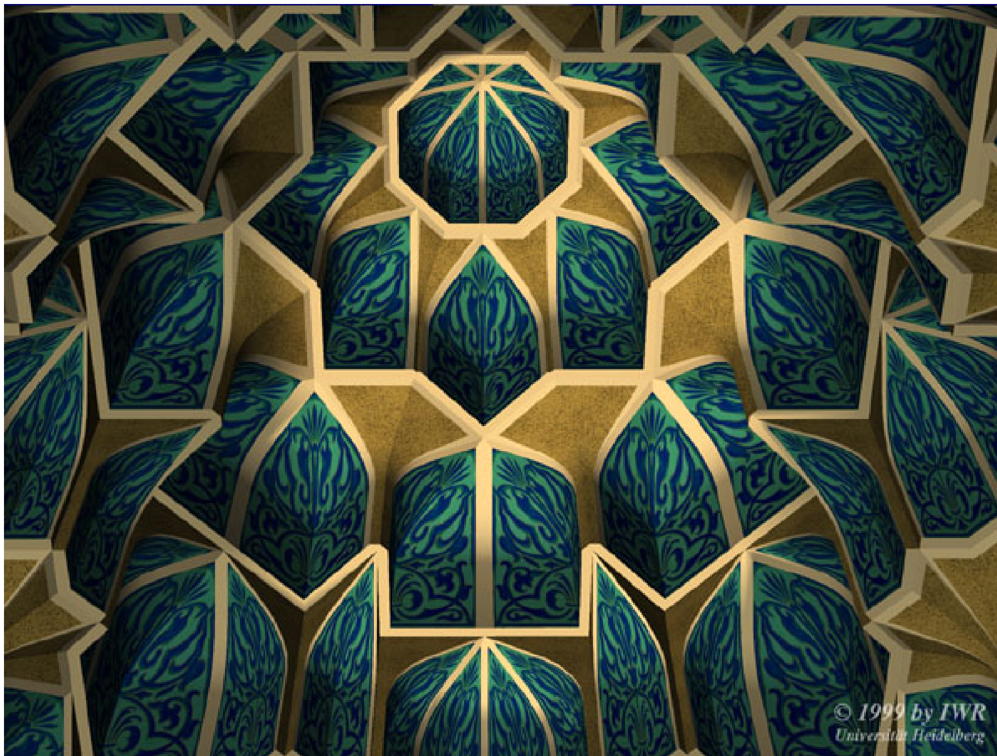


Abbildung 1.3. Mittels *OpenInventor* zusammengesetzte Muqarnas, arabische Kuppelausschmückung, Yvonne Dold, Christian Reichert, IWR, Universität Heidelberg

1965 *Computer Aided Design CAD* in der Flugzeugindustrie, Lockheed

1968 Tektronix Speicherröhre DUST erreicht Marktreife: Bei der Projektion eines Bildes auf die Röhre kommt es zur Teilentladung der lichtempfindlichen Speicherschicht. Ein Elektronenstrahl tastet im Zeilenrhythmus dieses Ladungsbild der lichtempfindlichen Speicherschicht ab. Die Energiemenge, die zur Wiederaufladung aufgebracht werden muss, lässt das elektrische Bildsignal entstehen.

1968 Gründung von *Evans und Sutherland*

1971 Gouraud Shading (Henri Gouraud, Schüler von Sutherland)

1974 Depth-Buffer Algorithmus (Edwin Catmull (Salt Lake City, Utah), Wolfgang Straßer (TU Berlin und Darmstadt, WSI/GRIS Tübingen))

1974 Anti-Aliasing (Herbert Freeman, Frank Crow (Salt Lake City, Utah))

1975 Phong Shading, Phong-Beleuchtungsmodelle (Phong Bui-Tuong), Normaleninterpolation

1980 Raytracing für Reflexionsberechnungen

1982 X-Windowsystem am MIT (erste Standards, GUS)

1984 *Silicon Graphics SGI* bringt das *Integrated Raster Imaging System IRIS* und die dazugehörige *Graphics Library GL* auf den Markt

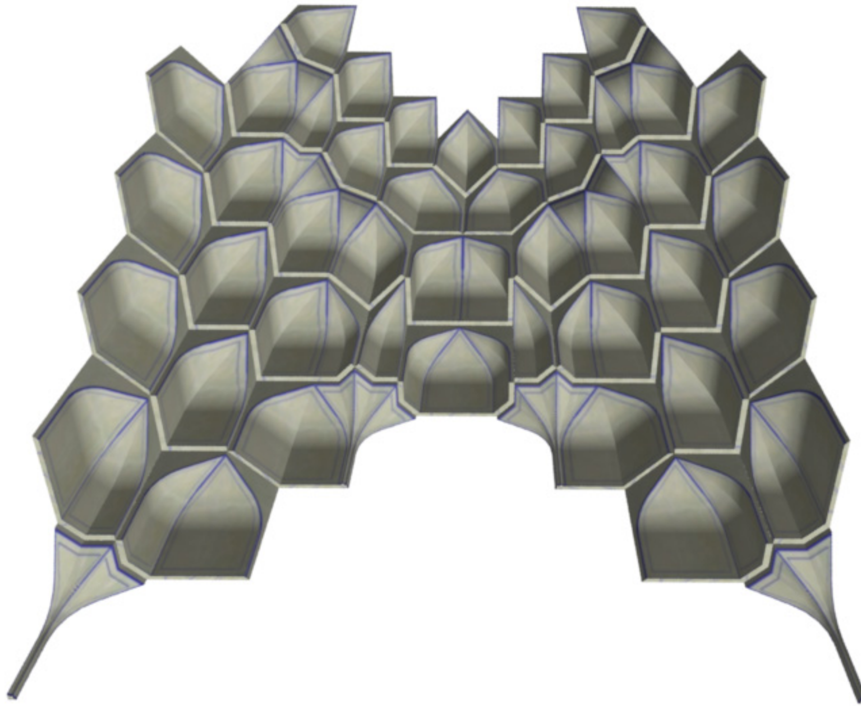


Abbildung 1.4. Aus 2D-Plänen automatisch rekonstruierte und mittels *Renderman* dargestellte Muqarnas, Silvia Harmsen, Daniel Jungblut, IWR, Universität Heidelberg

1984 Radiosity (Goral et al., Cornell University)

1985 *Graphisches Kernsystem GKS* [ISO85] (José Luís Encarnação, 1987 Gründung des Fraunhofer IGD Darmstadt)

1985 *Programmer's Hierarchichal Interactive Graphics System PHIGS*, ANSI-Standard mit 3D Erweiterung [ISO92]

1985 Softwarepatent für *Marching Cube Algorithm* (W.E. Lorensen, H.E. Cline), endete nach 20 Jahren am 5. Juni 2005

1986 Gründung von Pixar aus der *Digital Devision* von Lucas Film, Steve Jobs

1990 RendernMan Shading Language (Pixar), verwendet in Filmen wie *Toy Story* und *Finding Nemo*, (*The Renderman Companion*, Steve Upstill,1999)

1991 Effizientes Front-to-Back-Rendern mit *Binary-Space-Partitioning*, BSP-Trees (Dan Gordon, Lih-Shyang Chen)

1992 OpenGL 1.0, Cross-Platform 3d-Graphik Standard, Architecture Review Board (ARB)

1993 Hollywoods Filmindustrie greift Computeranimation für Filmpassagen auf *(Jurassic Park)*

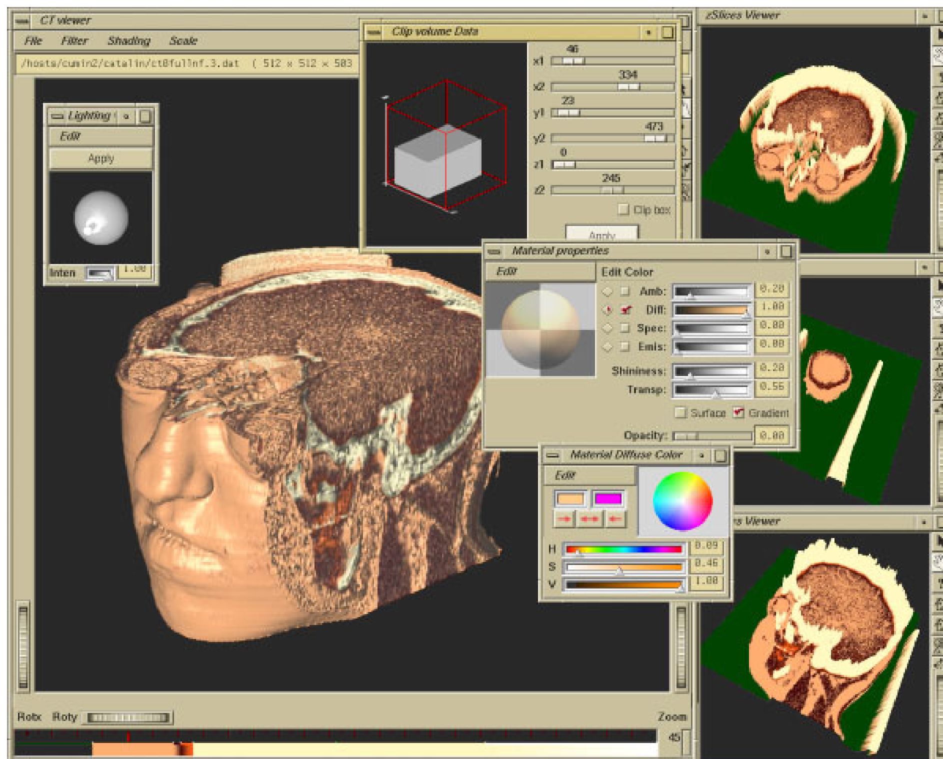


Abbildung 1.5. Graphisches User Interface (GUI) für das Volume Rendering Programm *vrend*, Catalin Dartu, IWR, Universität Heidelberg

1995 Mit *Toy Story* entsteht der erste vollständig als Computeranimation erstellte Kinofilm (John Lasseter, Walt Disney/Pixar). Es folgen 1998 die Filme *Antz* (Tim Johnson et al., Pacific Data Images (PDI)/Dream Works) und *A bug's life* (John Lasseter, Andrew Stanton, Walt Disney/Pixar)

1997 Subdivision surfaces als exakte Methode implementiert von Joe Stam, benutzt im Kurzfilm *Geri's Game*, Catmull-Clark, 1978

1998 Game Engines: Unreal Engine, Ego-Shooter Unreal

1998 Cool-to-Warm Shading, Nicht-Photorealismus (NPR) (Amy and Bruce Gooch, SIGGRAPH '98)

2000 *General Purpose Computation on Graphics Processing Unit* (GPGPU), programmierbare Pipeline durch Fragment- und Vertex-Shader zur Entlastung der CPU

2003 C for g Language, Cg-Tutorial von Mark J. Kilgard ist Bestseller auf der SIGGRAPH 2003, Meilenstein in der Graphikkartenprogrammierung

2004 OpenGL Shading Language, Randi J. Rost

2005 Physics Engine: 8. März 2005 wird *PhysX* auf der Game Developers Conference (GDC) vorgestellt. Computerspiele- und Simulationssoftware für physikalische Prozesse: *rigid body, collision detection, soft body, spring dynamics, rope and cloth, fluid dynamics*

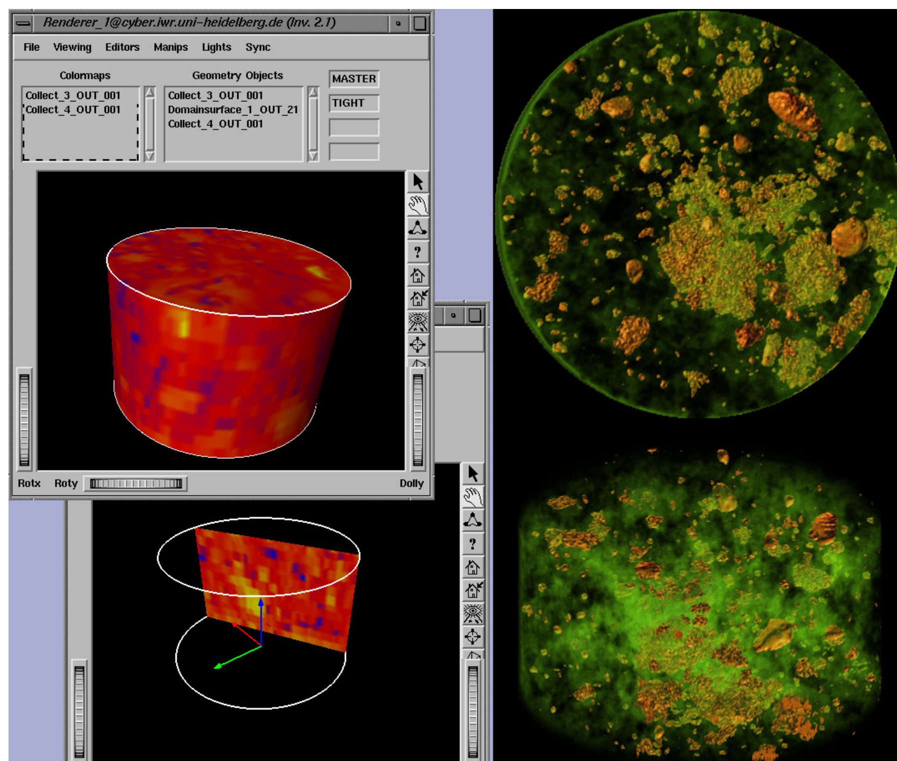


Abbildung 1.6. CT-Aufnahme eines Bohrkerns, Umweltphysik, Kurt Roth, IWR, Universität Heidelberg; links im *Direct Rendering, COVISE*, rechts mit *Volume Rendering*, vrend, Catalin Dartu, IWR, Universität Heidelberg

2007 Tesla Graphikkarte von nVidia OHNE Graphikausgang, parallele Rechenleistung mit *Compute Unified Device Architecture* (CUDA)

2008 Am 8. Dezember 2008 wird OpenCL von der Khronos Group, einem Zusammenschluss aller wichtigen Hardware-Hersteller, zu einem offenen Standard erklärt, mit dem alle Graphikkarten künftig programmierbar sein werden

2009 Im April 2009 sind erste Implementierungen von OpenCL verfügbar

1.4 Literaturhinweise

Die Literatur im Computergraphikbereich ist äußerst schnelllebig, da im Bereich der Spiele- und Filmindustrie eine rasche Entwicklung durch den großen Markt getrieben wird. Die meisten anspruchsvolleren Bücher und Zeitschriften sind bei Addison-Wesley erschienen.

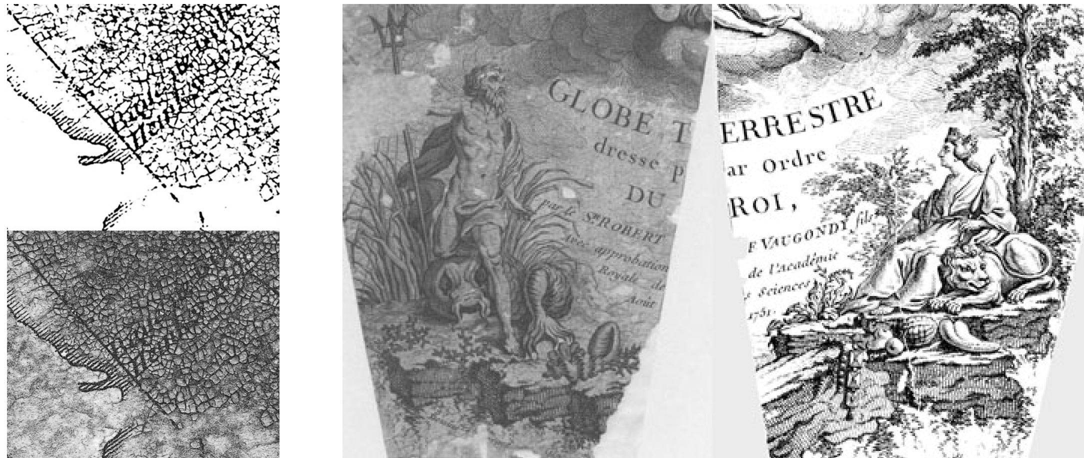


Abbildung 1.7. Nichtlineares Filterverfahren in der Bildverarbeitung, Restauration von Kupferstichen, Heidelberger Globus 1751, Alexander Dressel, Elfriede Friedmann, Susanne Krömker, IWR, Universität Heidelberg

1.4.1 Fachzeitschriften

IEEE - Computer Graphics and Applications will Theorie und Praxis miteinander verbinden. Sie erscheint sechsmal im Jahr, stellt spezielle Algorithmen bis hin zu vollen Systemimplementierungen vor und enthält auch Produktneuheiten. Kollumnist ist ein Altstar der Computergraphik, Jim Blinn.

ACM SIGGRAPH - Computer Graphics Quarterly ist die offizielle Zeitschrift der *Association for Computing Machinery's Special Interest Group on Graphics and Interactive Techniques* (ACM SIGGRAPH). Sie wird im Februar, Mai, August und November herausgegeben.

Springer-Verlag - Computing and Visualization in Science, G. Wittum, P. Deuflhard, W. Jäger, U. Maas, M. Rumpf (eds.), ist eine speziell aus dem Heidelberger Wissenschaftsumfeld gewachsene, noch junge Zeitschrift, wird bei Springer verlegt und behandelt spezieller die wissenschaftlichen Anwendungen.

1.4.2 Buchtipps

Das immer noch aktuelle Standardwerk zur Computergraphik ist von Foley et al., ist wiederholt überarbeitet worden und liegt sowohl als Einführung mit Grundlagenwissen [FvDF$^+$96b] als auch in seiner ausführlichen Variante [FvDF$^+$96a] vor.

Introduction to Computer Graphics J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, R. L. Phillips, Addison-Wesley, 1990, 1994, 1996 (2nd ed.)

Computer Graphics, Principles and Practice, J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, R. L. Phillips, Addison-Wesley, 1990, 1994, 1996 (2nd ed.)

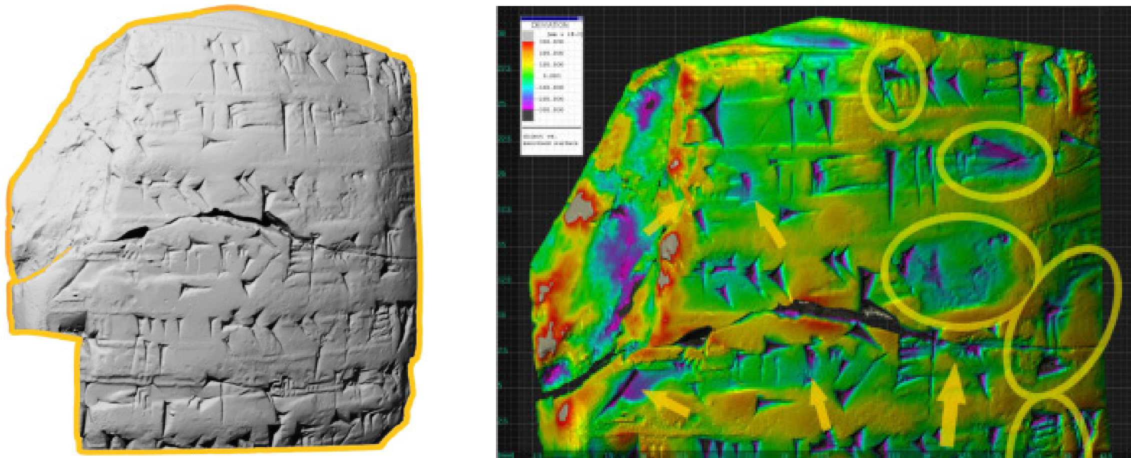


Abbildung 1.8. Mit einem 3D Scanner aufgenommene Keilschrift und die daraus erstellte Höhenkarte in Falschfarben, Hubert Mara, Bernd Breuckmann, IWR, Universität Heidelberg

Titel der deutschen Übersetzung: **Grundlagen der Computergraphik:** Einführung, Konzepte, Methoden, ebenfalls Addison-Wesley.

Aus ungefähr der gleichen Zeit ist das mit anschaulichen Zeichnungen zu den algorithmischen Erklärungen versehene grundlegende Buch [Wat90]:

Fundamentals of three-dimensional computer graphics, A. H. Watt, Addison-Wesley, 1989, 1990.

Ein besonders auf den mathematisch-algorithmischen Zusammenhang eingehendes, aktuelles Buch mit allerdings nur wenigen Skizzen und Farbtafeln ist bei Academic Press erschienen [Ebe01]:

3D game engine design, a practical approach to real-time computer graphics, D. H. Eberly, Academic Press, Morgan Kaufmann Publishers, 2001.

Das folgende Buch [SM00] ist ein in deutscher Sprache herausgegebener Titel, der allgemeiner auf das Thema wissenschaftlicher Darstellung von Daten ausgerichtet ist:

Visualisierung, Grundlagen und allgemeine Methoden, H. Schumann, W. Müller, Springer-Verlag Berlin Heidelberg, 2000.

Die vom Architecture Review Board herausgegebenen offiziellen OpenGL Bücher sind an ihren einheitlich gehaltenen Einbänden mit unterschiedlicher Grundfarbe zu erkennen. Das für den Einstieg in das Programmieren mit OpenGL unerläßliche ”Red Book” [SWND04a] liefert das nötige Hintergrundwissen in übersichtlichen Kapiteln nach Themen gegliedert. Es enthält zudem eigenständige Beispielprogramme zum Abtippen.

OpenGL programming guide: the official guide to learning OpenGl, version 1.4, OpenGL Architecture Review Board; Shreiner, Woo, Neider, Davis. 4th ed., Addison-Wesley, 2004.

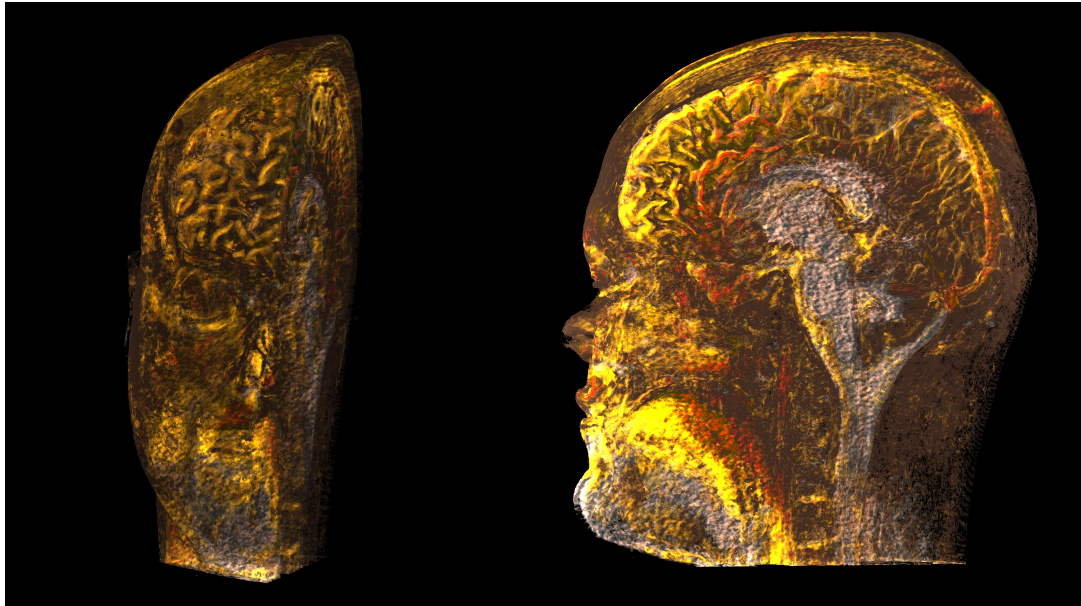


Abbildung 1.9. Implementierung eines Raycasting Algorithmus auf der Graphikkarte, Jens Fangerau, IWR, Universität Heidelberg (Daten aus der Arbeitsgruppe Prof. Meinzer, Deutsches Krebsforschungszentrum (DKFZ))

Das "Blue Book" [SWND04b] ist ein alphabetisch geordnetes Nachschlagewerk für gl-, glu- und glX-Aufrufe. Zudem enthält es eine sehr knappe Einführung und Zusammenfassung dessen, was mit diesen Aufrufen realisierbar ist.

OpenGL reference manual: the official reference document to OpenGL, version 1.4, OpenGL Architecture Review Board; Shreiner, Woo, Neider, Davis. 4th ed., Addison-Wesley, 2004.

Die Beschreibung der glut-Bibliothek [Kil96] ist mit grünem Einband versehen und übernimmt in vielen Programmen die Anbindung an das jeweilge Windowsystem. Sie gehört nicht zu den vom Architecture Review Board herausgegebenen offiziellen Werken und damit auch nicht zu einer Standard-Implementierung von OpenGL. Mittlerweile ist sie aber für die meisten Rechnerarchitekturen kostenlos verfügbar.

OpenGL Programming Guide for the X-Windowsystem, Mark J. Kilgard, Addison-Wesley, 1996.

1.5 Programmierhilfen

1.5.1 Kurzer Exkurs über `make`

Ein *Makefile* ist eine Utility mit vorgegebener Syntax. Sie wird mit `make <target>` aufgerufen und führt alle Kommandos aus, die sich hinter dem Doppelpunkt eines (ersten) `<target>` befinden. Das

Makefile selbst ist ein Textfile *ohne* besonderen *permissions mode* (r--------) (siehe auch 1.5.2). Sinn und Zweck des Shellkommandos make ist die *automatische Bestimmung der Programmteile, die erneut übersetzt werden müssen*, und die Bereitstellung der dazu nötigen *Kommandos*. Der Befehl make sucht im aktuellen Directory nach einer Datei makefile, danach Makefile, oder kann mit make -f mymake auch das File <mymake> ausführen. Innerhalb dieses Files müssen gewisse Syntaxregeln beachtet werden:

1. **Kommentare** beginnen mit #
2. **Macros** bestehen aus <STRING1> = [<string2>]
   wobei in <string2> alle Zeichen bis zu einem erneuten # oder nicht umsteuerten newline-Zeichen aufgenommen sind. Sollten Sie einen Zeilenumbruch erzeugen wollen, ohne dass der <string2> unterbrochen wird, beenden Sie die Zeile mit \. Der Aufruf der Macros geschieht über $(<STRING1>). (Konvention: Großbuchstaben für STRING1. Der <string2> darf auch leer sein!)
3. **Target Rules** bestehen aus <target> : [<prerequisit(s)>]
   und anschließenden Kommandos, arbeiten mit der Zeitmarke (modification time). Das bedeutet, dass das <target> immer dann neu erzeugt werden muß, wenn es älter ist, als sein(e) prerequisit(s) (falls welche angegeben sind). Außerdem muß eine bestimmte Syntax befolgt werden:

   ```
   <target> : [<prerequisit(s)>]
   <tab> <command1>
   <tab> <command2>
   <tab> <command3>
   <tab> <command4>
   ...
   ```
   Leerzeile

   Kommandos werden zeilenweise ausgelesen, der Reihe nach abgearbeitet und nach STDOUT geschrieben. Man trennt das nachfolgende Kommando entweder durch ein Semikolon oder durch eine neue Zeile, die mit einem Tabulator beginnt. Eine Leerzeile bedeutet, dass kein weiteres Kommando auszuführen ist.

   Double Suffix Rules sind spezielle *Target Rules*, wobei <target> aus .<suffix1>.<suffix2> besteht, e.g. .c.o, was bewirkt, dass über diese Regel aus Dateien mit der .<suffix1> solche mit der .<suffix2> erzeugt werden.

   ```
   .c.o:
   <tab> $(CC) $(CFLAGS) -c $<
   ```

   Diese beiden Zeilen bewirken, das aus my_file.c Objectcode mit dem Namen my_file.o durch das nach dem Tabulator vorhandene Kommando erzeugt wird. Dieses Kommando besteht im

Aufrufen des über ein Macro vereinbarten C-Compilers mit ebenfalls über ein Macro vereinbarten Optionen, wobei der Linkprozess unterdrückt und der Stammname `my_file` wieder benutzt wird.

`make <target>` führt ein bestimmtes `<target>` aus, wobei zunächst nach einer Datei

`makefile`

gesucht wird. Wird diese nicht gefunden, sucht das Programm nach `Makefile`. Findet sich keine solche Datei bzw. darin kein `target`, gibt das Betriebssystem auf (`don't know how to make <target>.`) Bemerkung: Es emphiehlt sich, `Makefile` mit großem `M` zu schreiben, da dies am Beginn eines Listings heraussticht. `make` ohne `<target>` führt das erste `<target>` aus, dass sich im `Makefile` findet.

Beispiel 1.1 *Dies ist ein kurzes Beispiel für ein typisches* `Makefile`:

```
#-----------------------------------------------------
#
# Makefile
#
# for simple openGl programs
# with GLUT-library for X-window and event handling
#
#-----------------------------------------------------


CC        = gcc
CCFLAGS   = -c
LIBS      = -L /usr/X11/lib
INCLUDES  =
LDFLAGS   = -lglut -lX11 -lGL -lGLU -lm

MYOBJECTS = A3.o
BIN       = A3


$(BIN): $(MYOBJECTS) Makefile
	$(CC) $(MYOBJECTS) $(LIBS) $(LDFLAGS) -o $(BIN)

.c.o:
	$(CC) $(INCLUDES) $(CCFLAGS) $<

clean:
	rm -f *.o $(BIN)
```

1.5.2 Noch kürzerer Exkurs über Shellskripte

Shellskripte sind ausführbare Textfiles und enthalten Unixbefehle, wie sie innerhalb der gewählten Shell gültig sind. Deshalb gehört in die erste Zeile immer der Aufruf der Shell (sonst entstehen leicht Probleme, wenn Sie Ihr Shellskript innerhalb einer anderen Shellumgebung benützen (lassen) wollen). Dieser Aufruf hat beispielsweise die Form `#!/bin/sh`, `#!/bin/tcsh` oder `#!/bin/bash`, wobei die nachfolgende Syntax nach den Regeln dieser Shell verfasst sein muss. Zum *Ausführen* tippt man den vereinbarten Namen der Skriptdatei, wobei der *Permissions mode* (wird gelistet mit dem *long listing* Kommando `ls -l` (oder `ll`), und gesetzt mit dem Befehl `chmod 700 <scriptname>`) auf `-rwx------` (read write execute, mit besonderem Nachdruck auf **execute**) stehen muss. Nachfolgende Argumente werden der Reihe nach an das Skript übergeben.

Beispiel 1.2 *Dies ist ein kurzes Beispiel für eine* Bourne Shell*, die eine C-Datei mit Suffix* `.c` *übersetzt und mit den Graphikbibliotheken verbindet:*

```
#!/bin/sh
if test $# -eq 0
then echo
  echo "This shell-script compiles a <filename>.c and links "
  echo "the X11, OpenGL and GLUT-Libs."
  echo "When you get this message, check for the <filename>."
  echo
  echo "Usage: my_cc (filename, without qualifier .c)"
else if test ! -s "$1.c"
  then echo "no file "$1.c" or "$1.c" is empty"
  else echo "cc "$1.c" -lglut -lGL -lGLU -lm "
    cc "$1.c" -o $1 -lglut -lGL -lGLU -lm
    echo "strip $1"
    strip $1
    $1
  fi
fi
```

Das `strip` *in diesem Skript bewirkt ein Verschlanken des* Executables.

1.6 OpenGL

OpenGl ist eine Entwicklung des *Architecture Review Board (ARB)*, das sich 1992 gegründet hat und dem im November 2004 auf dem *Promotor level* (stimmberechtigt mit jeweils einer Stimme) die Firmen 3DLabs, Apple, ATI, Dell Computer, IBM, Intel, nVIDIA, SGI und SUN Microsysterms angehören. Nichtstimmberechtigte Mitglieder *(Contributor level members)* sind die Firmen Evans &

Sutherland, Imagination Technologies, Matrox, Quantum 3D, S3 Graphics, Spinor GmbH, Tungsten Graphics, Xi Graphics.

Diese Firmen beschließen über Änderungen des Standards oder der Lizensierungen.

Das ARB definiert OpenGL folgendermaßen:

OpenGL ist ein Software-Interface zur Hardware. Zweck ist die Darstellung von zwei- und dreidimensionalen Objekten mittels eines Bildspeichers. Diese Objekte sind entweder *Images*, die aus *Pixeln* zusammengesetzt sind, oder *geometrische Objekte*, die durch *Vertices* (Raumpunkte) beschrieben werden.[1]

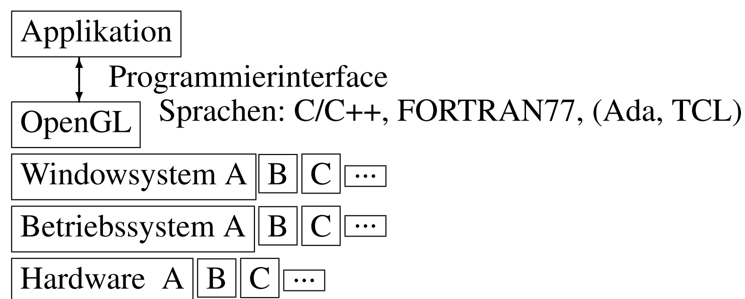


Abbildung 1.10. *OpenGL* bietet Sprach- und Architekturunabhängigkeit.

Das Programmierinterface ist sprachunabhängig, die gängigen Bibliotheken sind für C/C++ und FORTRAN, aber auch für Ada und Tcl erhältlich. Entwicklungen in der Graphiksoftware werden fast ausschließlich in C gemacht, aber auch FORTRAN kann dann sinnvoll eingesetzt werden, wenn bei Simulationen der größere Teil der Programme auf FORTRAN Quellcode beruht und die graphische Ausgabe der Ergebnisse einen geringen Teil des Programms umfasst. Zwar sind diese Sprachen auf Objektcodebasis miteinander zu linken, erfahrungsgemäß handelt man sich aber beim Portieren auf andere Rechnerarchitekturen immer wieder Fehler ein.

OpenGL bietet nicht die volle Funktionalität der IRIS GL, arbeitet dafür aber *unabhängig* vom Window- und Betriebssystem. Die Programmbibliothek OpenGL ist nur für die eigentliche Darstellung (Rendering) einer Szene mit (zwei- oder dreidimensionalen) Objekten ausgelegt.

1.6.1 Einbinden von Bibliotheksfunktionen

GL → Open**GL**, der Graphikstandard und Name der nötigen *Directories*. Steht der Aufruf

```
#include <GL/gl.h>
```

[1] aus: Graphikprogrammierung mit OpenGL, Barth, Beier, Pahnke, Addison-Wesley, 1996

in ihrem Programm, wird unter einem Standardpfad für `INCLUDES` nach der Headerdatei `gl.h`, also z.B. `/usr/include/GL/gl.h` gesucht. Beim Linken ihrer einzelnen Programmteile fügen Sie mit der Option

```
-lGL
```

die Bibliothek `/usr/lib/libGL.a` bzw. `/usr/lib/libGL.so` (shared library) aus dem Standardpfad hinzu.

1.6.2 Zusatzbibliotheken

Aufgrund der für das modulare Aufbauen nötigen Einschränkung braucht man weitere Bibliotheken:

GLU → Open**GL** **U**tility Library, gehört zu jeder OpenGL-Implementierung, die Headerdatei wird mit

```
#include <GL/glu.h>
```

ins Programm aufgenommen, wobei sich die Zeile `#include <GL/gl.h>` jetzt erübrigt, sie ist nämlich in `glu.h` enthalten.

```
-lGLU
```

ist die Option zum Linken der Bibliothek aus dem Standardpfad `/usr/lib/libGLU.a` bzw. `libGLU.so`. Diese *Utilities* umfassen gegenüber den Kernfunktionen, die alle über `libGL.a` erreichbar sind,

- Vereinfachte geometrische Funktionen
- High-level Primitive (Kugel, Zylinder, Torus)
- *Nonuniform rational B-Splines NURBS*
- Polygon Tesselation
- Skalieren von *Images*

- Generieren von Texturen aus *Images*
- Fehlermeldungen

Die Einbindung in diverse Fenstersysteme gehört **nicht** zur OpenGL-Implementierung! Man kann sich selbst um die X11 Anbindung kümmern: Verwaltungsfunktionen sind in der **GLX** implementiert:

- Eigenschaften des X-Servers
- *Offscreen Rendering*
- *Event Handling*
- Bildspeicher
- Schriftsätze

Die Fensteranbindung unter *WindowsNT* heißt **WGL**, unter *OS/2* von IBM heißt sie **PGL** (stirbt gerade aus, Support bis 2004). Das von IBM entwickelte Betriebssystem AIX, ein UNIX-derivat, hat X11-Anbindung.

Wenn man sich nicht tiefer mit diesen Anbindungen beschäftigen möchte, benutzt man die

GLUT → Open**GL** **U**tility Toolkit von Mark J. Kilgard, die eine Headerdatei

```
#include <GL/glut.h>
```

benötigt, die schon das Einbinden der Header `gl.h` und `glu.h` enthält.

```
-lglut
```

ist die Option zum Linken der Bibliothek aus dem Standardpfad `/usr/lib/libglut.a`.
Außer den unter **GLX** genannten Aufgaben übernimmt die `libglut.a`

- Fensterverwaltung
- *Callback-Mechanismen, wie* Keyboard- *und* Mouse-*Anbindung*
- *Timer*-Prozeduren

- Einfache Menüsteuerung
- *Highest-Level* Primitive, z.B. `glutWireTeapot(GLdouble size);`

Beispiel 1.3 *OpenGL und GLUT*

```
/**********************************************
  Solution A3, Computergraphics I, SS 2001
  Author:  Susanne Kroemker, IWR - UNIVERSITAET HEIDELBERG
  phone:   +49 (0)6221 54 8883
  email:   kroemker@iwr.uni-heidelberg.de
**********************************************/
#include <GL/glut.h>
#include <stdio.h>

void winPrint (char *);
void winPrintPos (int, int, int, char *);
void display(void);
int main(int, char **);

void winPrintPos (int x, int y, char *str)
{
  glRasterPos2i(x,y);
  for (j=0; j<strlen(str); j++)
    glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18,str[j]);
}

void display(void)
{
  char *words = "Hello World";

  glClear (GL_COLOR_BUFFER_BIT);
  glColor3f (1.0,0.0,0.0);

  glBegin(GL_POLYGON);
    glVertex2i (0, 20);
    glVertex2i (90, 70);
    glVertex2i (100, 85);
    glVertex2i (90, 100);
    glVertex2i (55, 105);
    glVertex2i (25, 100);
    glVertex2i (0, 80);
  glEnd();
  glBegin(GL_POLYGON);
    glVertex2i (0, 20);
    glVertex2i (-90, 70);
    glVertex2i (-100, 85);
    glVertex2i (-90, 100);
    glVertex2i (-55, 105);
    glVertex2i (-25, 100);
    glVertex2i (0, 80);
```

```
  glEnd();

  glColor3f (0.7,0.7,1.0);
  winPrintPos(-20,70,words);

  glFlush();
}

int main(int argc,char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
  glutInitWindowSize (250,250);
  glutInitWindowPosition (100, 100);
  glutCreateWindow ("Hello World");
  glClearColor (1.0, 1.0, 1.0, 0.0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(-120.0, 120.0, 15.0, 120.0);
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```

1.6.3 Namenskonventionen

Um die Bibliotheksaufrufe im Quellkode leicht von den selbstdefinierten Unterprogrammen unterscheiden zu können[2], hat man die in Tabelle 1.1 notierten Namenskonventionen eingeführt.

Präfix	Bedeutung
gl	Alle OpenGL-Funktionen beginnen grundsätzlich mit diesem Präfix, danach schließt ein Großbuchstabe an, dann (bis zum nächsten Wortanfang) wird klein weiternotiert
glu	Alle Aufrufe aus der Utility Bibliothek
glut	Alle Aufrufe aus der Utility Toolkit Bibliothek
GL_	vordefinierte Konstanten der Headerdateien, Flags zum Steuern von Modi

Tabelle 1.1. Konventionen der Bibliotheksfunktionen und Headerdateien

Genauso gibt es Vereinbarungen bei Datentypen:

[2] In der älteren GL (Graphics Library), für dortige GL-Aufrufe, ist das nicht der Fall.

Datentyp	Suffix	OpenGLtyp	bits
signed char	b	GLbyte	8
short	s	GLshort	16
int	i	GLint	32
float	f	GLfloat	32
double	d	GLdouble	64
unsigned char	ub	GLubyte	8
unsigned short	us	GLushort	16
unsigned int	ui	GLuint	32

Tabelle 1.2. Spezifikation der GLtypischen Bitlängen

Implementierungen unter FORTRAN und C benötigen diese Spezifizierungen, unter C++ und Ada sind sie obsolet.

Beispiel 1.4 *OpenGL-Aufrufe, um einen Punkt* $(x, y)^T = (1, 3)^T$ *anzusteuern*

```
glVertex2i(1,3);
glVertex2f(1.0,3.0);
```

```
GLfloat vertex_vector[]={1.0,3.0};
glVertex2fv(vertex_vector);
```

1.6.4 OpenGL ist eine *State machine*

Innerhalb des Programmablaufs wird ein bestimmter Zustand oder Modus eingestellt, der solange aktiv bleibt, bis er wiederum geändert wird. Jede Zustandsvariable hat eine Voreinstellung (*default value*); An jeder Stelle des Programmablaufs lässt sich wahlweise mit

```
glGetBooleanv();
glIsEnabled();
```

dieser Zustand abfragen.

Beispiel 1.5 *Die Farbe ist eine Zustandsvariable*

```
glColor3f(1.0,0.0,0.0);
```

Im RGB-Modus ist die Intensität jeweils zwischen 0 und 1 gespeichert. Mit obigem Befehl ist der rote Farbkanal maximal, der grüne und blaue Null gesetzt, und das Ergebnis ist der rote Farbzustand.

ACHTUNG! *Wenn Sie mit roter Schrift in rote Herzen malen, werden Sie sehen, dass Sie nichts sehen.*

Weitere Zustandsvariablen sind

- Ansichts- und Projektionstransformationen
- *Polygon Drawing Modes*
- *Pixel-Packing* Konventionen (Auslesen des Bildschirmspeichers)
- Positionen und Eigenschaften des Lichts
- Materialeigenschaften

Beispiel 1.6 *Viele Zustände, wie beispielsweise Licht und einzelne Lichtquellen können einfach ein- und ausgeschaltet werden. Welche Zustände mit den Schalterfunktionen bedient werden, regeln vordefinierte Konstanten, die in den Headerdateien ihre Werte zugewiesen bekommen. Beim Licht entspricht* `GL_LIGHTING` *dem Hauptschalter für alle Lichtquellen,* `GL_LIGHT0, GL_LIGHT1` *bis zur maximalen Anzahl* `GL_MAX_LIGHTS` *entsprechen einzelnen Lichtschaltern.*

```
glEnable(GL_LIGHT0);
.
glDisable(GL_LIGHT0);
```

Aber auch andere aufwendige Tests können aus dem Ablauf eines Programms mit `glDisable(..)` *herausgenommen werden, beispielsweise* `GL_DEPTH_TEST`*.*

1.7 Graphikpipeline

Geometrische Daten bestehen zunächst aus Punkten, die mit Linien verbunden sein können, und schließlich Dreiecke sowie komplexere Polygone oder Polyeder formen. Es sind von ihrer Ausdehnung 0D, 1D, 2D und 3D-Objekte, die in einem dreidimensionalen virtuellen Raum eingebettet sind. **Bilddaten** sind sogenannte *Images* oder *Bitmaps*, die entsprechend dem Bildschirmspeicher zeilenweise Farbinformationen (oder Intensitäten) enthalten. Beide Datentypen werden während der Bildgenerierung unterschiedlich gehandhabt, durchlaufen aber schließlich dieselben Endoperationen, bevor die endgültigen Pixelinformationen in den Bildschirmspeicher geschrieben werden.

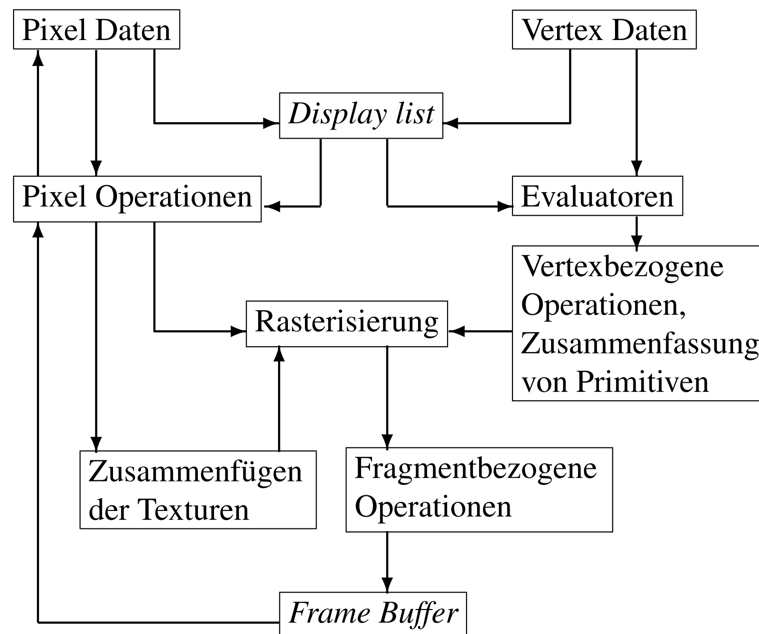


Abbildung 1.11. ***Rendering Pipeline* in *OpenGL*.**

Am Ende dieses Kapitels sollten Sie sich nochmals klarmachen, dass das Erzeugen von Bildern ein mehrstufiger Prozess ist. Er umfasst zunächst die *Modellierung*, danach das sogenannte *Rendering* und schließlich die Bildausgabe.

Modellierung: Eingabedaten werden einem *Preprocessing* unterzogen

- **Projektionen** hochdimensionaler vektorwertiger Daten
- **Skalierungen** einzelner skalarer Komponenten auf zueinander passende Größenordnungen
- **Datenreduktion** durch Ausdünnen

Diese Vorbereitung führt zu 3D-Datenpunkten und daraus zusammengesetzten Objekten.

Rendering: Datennachbereitung oder *Postprocessing*

- Transformationen
- Abbildungen
- Sichtbarkeit (Clipping, Transparenz)
- Lichtführung, Beleuchtung
- Rasterisierung

Diese Datenaufbereitung führt zu 2D-Imagedaten, die jetzt sichtbar gemacht werden sollen.

Bildausgabe: Bildschirm(speicher) oder *Framebuffer* sowie weitere Ausgabegeräte, die den Bildschirmspeicher auslesen, oder direkte Ausgabe durch *Offscreen Rendering* an

- Kamera (vor hochauflösendem Schwarz-Weiß-Schirm)
- Drucker mit eigenem *PostScript Interpreter*
- Video (mit geringerer Auflösung, niedrigeren Bandbreiten)
- Dateien verschiedener Formate und Komprimierungen

Die Ausgabe führt bei unsachgemäßer Ansteuerung zu deutlichen Qualitätsverlusten.

1.8 Übungsaufgaben

Aufgabe 1.1 Makefiles, C–Compiler

Erstellen Sie ein einfaches C-Programm, das in einer Schleife 10x „Hello World!" auf dem Bildschirm ausgibt. Compilieren Sie dieses Programm mit dem vorhandenen C–Compiler `cc`*. Der dazu benötigte Befehl lautet:*

```
cc FILE [-o EXECUTABLE]
```

Schreiben Sie ein File `makefile`*, das Ihnen bei Eingabe des Befehls* `make all` *genau diese Aufgabe abnimmt. Definieren Sie dazu im Makefile u.a. die Makefile–Makros* `CC` *und* `CFLAGS`*, um je nach Umgebung, Rechner und Betriebssystem Compiler und Flags frei wählen zu können. Erzeugen Sie einen weiteren Eintrag im* `makefile`*, der Ihnen bei Eingabe von* `make clean` *das Executable und alle eventuell vorhandenen Objekt–Files löscht. Modifizieren Sie ihr Makefile so, dass zuerst alle C–Files Ihres Projektes (hier also nur eines) separat compiliert werden und danach einmalig der Linker zur Programmerstellung aufgerufen wird. Erweitern Sie Ihr Makefile danach um die Makefile-Makros* `INCPATH`*,* `LIBPATH` *und* `LIBS` *für den Include–Pfad, den Library–Pfad und die verwendeten Libraries.*

Aufgabe 1.2 Zerlegen in Einzeldateien

a) Zerlegen Sie Ihr Programm in mindestens drei Dateien, die Sie mit einem Makefile übersetzen (z. B. `main.c`*,* `displayFunction.c` *und* `controlFunctions.c`*, wobei Sie auf die Transformationsroutinen und die Menüsteuerung aus früheren Programmen und Musterlösungen zurückgreifen).*

b) Binden Sie eine eigene Headerdatei z. B. `myHeader.h` *ein, in die Sie wiederkehrende* `#include` *und* `#define` *Aufrufe sowie Prototype descriptions und globale Variablen unterbringen.*

Aufgabe 1.3 Zufallszahlen

Viele Computergrafik-Algorithmen verwenden Zufallszahlen, um ihre Resultate zu erzielen. Dabei ist es wegen der grossen Menge benötigter Zufallszahlen (z.B. beim Verrauschen von gerenderten Bildern, bei der Erstellung fraktaler Landschaften etc.) wichtig, einen guten Zufallszahlengenerator zu verwenden.

Zu den Gütekriterien gehört dabei, dass die entstehenden Zahlen im vorgegebenen Intervall statistisch gleichverteilt sind und dass aufeinanderfolgende Paare von Zufallszahlen nicht korreliert sind.

1. *Erstellen Sie unter C einen Vektor von 10.000 Zufallszahlen im Intervall* $[0, 1]$. *Ermitteln Sie danach für eine Einteilung des Einheitsintervalls in 20 gleichlange Teilintervalle, wie sich die Zufallszahlen auf diese Intervalle verteilen. Schreiben Sie das Ergebnis in der Form*

   ```
   0.00  0.05    543
   0.05  0.10    489
   0.10  0.15    502
   ...
   ```

   in ein File und stellen Sie das Ergebnis als geeignete Grafik in Gnuplot dar.

2. *Erstellen Sie unter C ein Feld von 4.000 Zahlenpaaren, deren Koordinaten Sie direkt nacheinander vom Zufallsgenerator auswählen lassen. Schreiben Sie die Paare in ein File, lesen Sie es in Gnuplot ein und plotten Sie die 4.000 Datenpunkte in* $[0, 1] \times [0, 1]$.

Beurteilen Sie anhand der Ergebnisse die Güte des Zufallszahlengenerators.

Aufgabe 1.4 'Hello World'

Schreiben Sie ein C-Programm, dass unter Verwendung von den Bibliotheken OpenGL und GLUT ein Fenster öffnet und dahinein ein beliebiges konvexes Polygon in 2D zeichnet. Schreiben Sie mit Hilfe der Bitmap–Funktionalitäten von OpenGL zwei Worte in das Polygon ('Hello World').

Kapitel 2

Koordinatensysteme

Das kartesische Koordinatensystem in 2D, die *Zeichenfläche* (Papier, Bildschirm, Plotter) hat seinen Ursprung *links unten* (oder mittig auf dem Blatt).

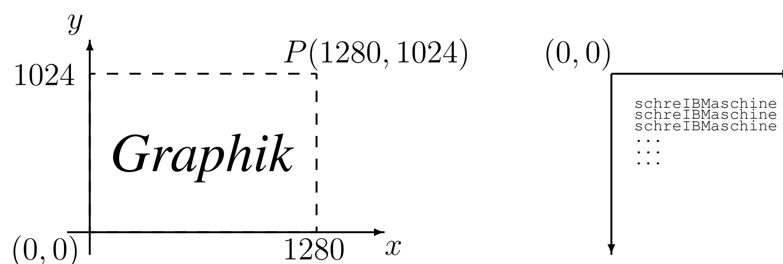


Abbildung 2.1. Koordinatenursprung und positive Richtungen.

Die *Schreibfläche* (Textverarbeitungsprogramme) und deshalb auch heute noch viele PCs (IBM - Schre**ibm**aschine) haben den Nullpunkt *links oben*, so dass im kartesischen Koordinatensystem der PC-Ursprung die Koordinaten $(0,\ y_{max})$ hat. Es ist eine Transformation nötig, die die Punkte des kartesischen Systems $(x,\ y)$ in die Punkte des kartesischen Systems $(x,\ y_{max} - y)$ überführt.

Damit man sich nicht an den *Gerätekoordinaten* (hier maximale Pixelanzahl des Schirms) ausrichten muß, sind *Skalierungen* oder *Translationen* nötig. *Drehungen* gewinnen mit der 3D-Darstellung zusätzliche Bedeutung.

2.1 Welt-, Bild- und Gerätekoordinaten

Koordinatensysteme können nach unterschiedlichen Kriterien definiert werden:

1. 1D, 2D, 3D, ...
2. kartesisch, polar, zylindrisch
3. absolut, relativ
4. fest, temporär
5. systembezogen, benutzerdefiniert

Das Weltkoordinatensystem (WKS) ist ein Koordinatensystem, das allen anderen Koordinatensystemen zugrunde liegt. Es muss auf die Koordinaten des Ausgabegeräts abgebildet werden. Dazu wird ein Fenster geöffnet, dessen Größe und Position auf Gerätekoordinaten umgerechnet wird. Das bekannteste Fenstersystem, das nach dem Client-Server Modell funktioniert, ist das am MIT 1984 entwickelte X-Window System, das als *X version 11* (oder X11) aus dem Jahr 1987 bis heute verwendet wird.

Abbildung 2.2. X version 11, X11

Window - Weltkoordinaten, Skalierung

Viewport - Gerätekoordinaten, Fenster auf dem Schirm, Ausmaße

Der Viewport ist ein Ausschnitt aus der 3D-Welt; dieser Ausschnitt soll später im Fenster dargestellt werden. Er ist das Sichtfeld. Für die Voreinstellung (default) des Viewports gilt, dass er genauso groß wie das Fenster ist. Eselsbrücke: Denken Sie bei einem Viewport an einen Fensterausschnitt.

Die Bibliotheksaufrufe in OpenGL bzw. glut lauten:

```
glutInitWindowSize(GLint width, GLint height);
```
Angaben in Bildschirmkoordinaten = Pixelkoordinaten

```
glutInitWindowPosition(GLint x, GLint y);
```
Abhängig vom Windowsystem, bezeichnen die Koordinaten (meist) die obere linke Ecke, wobei die Fensterrahmen zwingend auf den Schirm abgebildet werden und bei einer Angabe $(x, y) = (0, 0)$ eine Verschiebung um die jeweilige Rahmenbeite nach rechts unten bedingen.

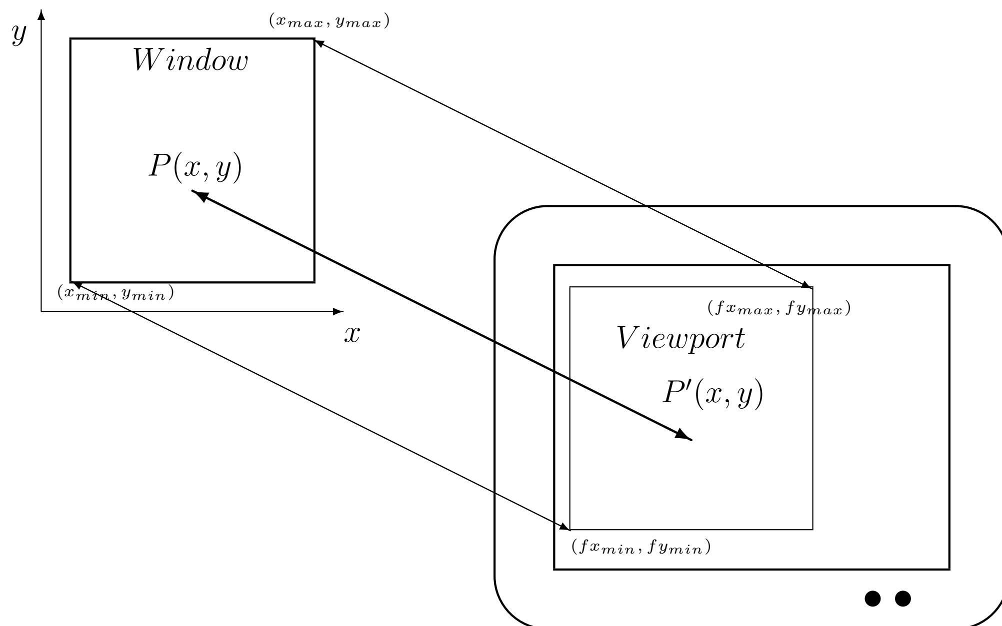


Abbildung 2.3. *Window* und *Viewport*.

```
glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

Beim Viewport werden linke untere Ecke und Größe gleichzeitig angegeben. Per default entsprechen die Angaben den Fensterangaben.

2.2 Transformationen

Zunächst betrachten wir affine Transformationen in 2D. Ein Punkt $P(x, y)$ wird durch den Vektor $(x, y)^T$ eindeutig festgelegt. Eine affine Transformation ist eine Abbildung zwischen zwei Vektorräumen, die Kollinearitäten und Abstandsverhältnisse paralleler Strecken bewahrt. Sie ist

- bijektiv (eineindeutig)
- geradentreu und
- das Teilverhältnis von Strecken ist invariant.

Daraus ergibt sich, dass

- nichtentartete Dreiecke auf beliebige nichtentartete Dreiecke abgebildet werden,
- parallele Geraden auf parallele Geraden abgebildet werden, und dass

- das Flächenverhältnis von Original und Bild gleich der Determinante ($\neq 0$) der Abbildungsmatrix ist.

Eine 2D-Abbildung, bestehend aus einer *Drehung* und einer *Translation*, hat die allgemeine Form

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}.$$

Eine *Ähnlichkeitsabbildung* ist eine affine Abbildung mit der Eigenschaft, dass das Verhältnis von Bildstrecke zu Originalstrecke eine Konstante, das *Ähnlichkeitsverhältnis* k, ist.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} k \cos\beta & \pm k \sin\beta \\ k \sin\beta & \pm k \cos\beta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}.$$

Hat das Verhältnis den Betrag 1 und handelt es sich um eine im mathematischen Sinne positive Drehung um den Winkel β, so lautet die Abbildungsmatrix

$$\begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix}.$$

Beispiel 2.1 Vertauschter Koordinatenursprung
Spiegelung an der Geraden parallel zur x-Achse mit $y = y_{max}/2$. Erwartetes Resultat:

$$x' = x \qquad y' = y_{max} - y$$

Abbildungsgeometrisch:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ y_{max} \end{pmatrix}.$$

Beispiel 2.2 Schriftzeichen *(typischerweise 2D)*
Schriftzeichen werden vom Plotter (Drucker/Bildschirm) *als Folge von Vektoren definiert, d.h. jedes Zeichen wird als eine Liste von Koordinaten vorgegeben, die nacheinander vom* Stift *angefahren werden (Malen nach Zahlen* ;-) *). Im folgenden Beispiel ist das*

Drehen:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos\beta & -sin\beta \\ sin\beta & cos\beta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Dehnen (Breite b, Höhe h):

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} b & 0 \\ 0 & h \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Kursivieren:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & n \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

ein Hintereinanderschalten von Matrixoperationen (Drehwinkel $\beta = 45^o$, Breite $b = 10$, Höhe $h = 15$, Neigung $n = 0.25$):

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 7.071 & -0.176 \\ 10.607 & 10.607 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Für das Schreiben in Graphikfenster stellt die GLUT-Bibliothek Fonts zur Verfügung:

```
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, str[.]);
```

Innerhalb von OpenGL wird nur die Rasterposition angesteuert:

```
glRasterPos2i(GLint x, GLint y);
```

2.3 Homogene Koordinaten

Der Nachteil der über Matrixmultiplikationen erhaltenen Abbildungen besteht darin, dass sie immer einen Fixpunkt im Ursprung besitzen.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Will man über Matrixmultiplikationen auch die Translation (oder andere Transformationen, bei denen sich alle Punkte ändern) erfassen, muss der Fixpunkt außerhalb des Ursprungs liegen. Dazu führt man eine weitere Dimension ein (für 2D eine z-Koordinate, für 3D eine w-Koordinate) und legt für die Bildebene $z = 1$ ($w = 1$) fest. Der Fixpunkt der Matrixmultiplikation liegt damit *außerhalb der Bildebene!*

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Drehen, Skalieren und Spiegeln bedeutet $a_{13} = a_{23} = 0$ und ist trivial in z (w). Translationen um U in x-Richtung und V in y-Richtung schreiben sich als

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & U \\ 0 & 1 & V \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x+U \\ y+V \\ 1 \end{pmatrix}.$$

Hintereinanderschalten von Transformationen in diesen *Homogenen Koordinaten* ist durch wiederholte Matrixmultiplikation möglich.

Bemerkung 2.1 ACHTUNG! *Matrixmultiplikationen sind* NICHT KOMMUTATIV! *Anschaulich gesprochen: Wenn man ein im Ursprung befindliches Objekt zuerst dreht und dann verschiebt, sieht das Resultat völlig anders aus, als wenn man es zuerst aus dem Ursprung heraus verschiebt und dann um den Ursprung dreht.*

2.3.1 OpenGL arbeitet mit homogenen Koordinaten ($w = 1$)

Die dreidimensionale *Projektive Geometrie* zeichnet eine vierte Koordinate w aus, die innerhalb von OpenGL spezifiziert werden kann, was aber eigentlich nicht gemacht wird. Üblicherweise ist $w = 1$ und es ergibt sich die natürliche Identifizierung mit dem dreidimensionalen *Euklidischen Raum* über $(x/w, y/w, z/w)^T$. 2D-Koordinaten $(x, y)^T$ werden als $(x, y, 0.0, 1.0)^T$ behandelt.

Der Punkt $w = 0$ ist der *unendlich ferne Punkt* der *Projektiven Geometrie*, bei der Punkte mit Ursprungsgeraden identifiziert werden.

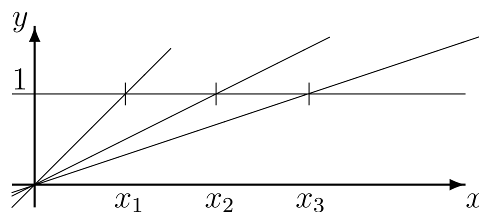


Abbildung 2.4. Projektive Geometrie identifiziert Ursprungsgeraden und Punkte.

2.4 Kameraanalogie

Bisher haben wir homogene Objektkoordinaten und Matrizen zu Abbildungszwecken kennengelernt.

$$\underbrace{\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}}_{\text{Vertex}} \rightarrow \underbrace{\begin{pmatrix} \texttt{Model-} \\ \texttt{view-} \\ \texttt{Matrix} \end{pmatrix}}_{\text{Eye-Coord.}} \rightarrow \underbrace{\begin{pmatrix} \texttt{Pro-} \\ \texttt{jection-} \\ \texttt{Matrix} \end{pmatrix}}_{\text{Clip-Coord.}} \rightarrow$$

$$\underbrace{\begin{pmatrix} \texttt{Pers-} \\ \texttt{pective} \\ \texttt{Devision} \end{pmatrix}}_{\text{Norm. Device Coord.}} \rightarrow \begin{pmatrix} \texttt{Viewport-} \\ \texttt{Trans-} \\ \texttt{formation} \end{pmatrix} \rightarrow \begin{pmatrix} \texttt{Window-} \\ \texttt{Coord.} \end{pmatrix}$$

Bemerkung 2.2 *Der Viewport spezifiziert die affine Transformation von* Device *Koordinaten auf die Fensterkoordinaten.*

Wie man von Objektkoordinaten zu Fensterkoordinaten kommt, lässt sich am ehesten in der Begriffswelt der Fotografie erklären.

2.4.1 Aufstellen der Kamera

Der Betrachter einer virtuellen 3D-Welt ist mit der Kameraposition identisch. Allerdings ist es manchmal sinnvoll, die Position der Kamera in Bezug auf eine komplexe Szene darzustellen, beispielsweise, wenn man eine Kamerafahrt durch eine virtuelle Welt plant. In solchen Fällen wird man ein Symbol für die Kameraposition entwickeln und sich als Betrachter aus größerer Entfernung die komplette Szene anschauen. Meist platziert man daneben ein weiteres Fenster, in dem man den Kamera-Ausschnitt sieht: Der Betrachter einer virtuellen 3D-Welt ist mit der Kameraposition identisch. *Viewing Transformation* heißt, die Blickrichtung zu verändern.

2.4.2 Arrangieren der Szene

Modelling Transformation betrifft das Verschieben, Drehen und Skalieren des Objekts. *ACHTUNG!* Wenn Sie die Augenposition dem Objekt nähern, hat das den gleichen Effekt wie das Objekt den Augen (der Kamera) zu nähern oder das Objekt in jede Raumrichtung zu vergrößern. *Viewing* und *Modelling Transformations* werden daher in der *Modelviewmatrix* gespeichert.

2.4.3 Linseneinstellung

Eine *Projection Transformation* entspricht der Veränderung, die man an der Kameralinse vornehmen kann. Eine *Orthographische Projektion* erzeugt eine rechtwinklige Parallelprojektion. Längen bleiben unabhängig von der Tiefe des Raumes erhalten.

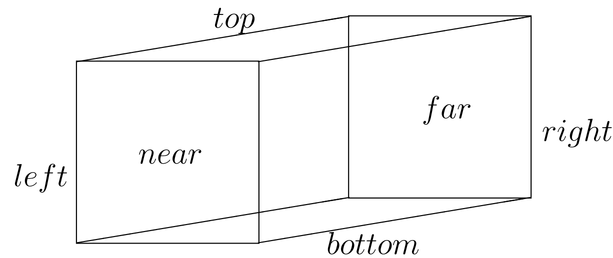


Abbildung 2.5. Orthographische oder Parallelprojektion.

```
glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble
zNear, GLdouble zFar);
```

Dabei wird `(left, bottom, -zNear)` auf die linke untere Ecke und `(right, top, -zNear)` auf die rechte obere Ecke des Viewports abgebildet und stellt damit die *Vordere Begrenzungsebene (near clipping plane)* dar. Entsprechend gibt es eine *Hintere Begrenzungsebene (far clipping plane)* bei `-zFar`, bis zu der die 3D Objekte dargestellt werden. Wenn Sie Ihr Objekt aus diesem Bereich hinausschieben, wird es zunächst angeschnitten und verschwindet schließlich.

Damit die Transformationsmatrix nicht singulär wird muß `zNear` $\neq$ `zFar` sein.

Für den einfachen Fall, dass Sie ein 2D Bild auf den Schirm bringen wollen, benutzen Sie

```
gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);
```

was dem obigen `glOrtho();` mit der Voreinstellung `zNear = -1, zFar = 1` entspricht. Die 2D-Vertices werden mit z-Wert 0 angenommen, so dass alle Objekte zwischen linker und rechter, oberer und unterer Begrenzung dargestellt werden können (no clipping!).

Der Aufruf von `glOrtho();` entspricht einer Multiplikation der aktuellen Matrix mit der Transformationsmatrix

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & \frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & \frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{zFar-zNear} & \frac{zFar+zNear}{zFar-zNear} \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

wobei die Augenposition in $(0, 0, 0)$ angenommen wird, und `zNear, zFar` beide positive oder negative Werte haben können.

Perspektivische Projektion wird mit dem OpenGL-Aufruf

```
glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble
```

```
zNear, GLdouble zFar);
```

gemacht (Frustum heißt Kegelstumpf), wobei die Transformationsmatrix

$$\begin{pmatrix} \frac{2zNear}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2zNear}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{zFar+zNear}{zFar-zNear} & -\frac{2zFarzNear}{zFar-zNear} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

lautet. Hier gilt nun, dass `zFar > zNear > 0` beide echt positiv sein müssen (ansonsten wird ein `GL_INVALID_VALUE` erzeugt, Fehlermeldung).

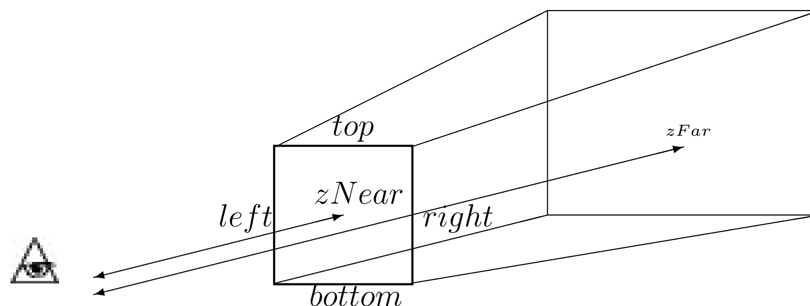


Abbildung 2.6. Perspektivische Projektion, der Kegelstumpf.

Etwas komfortabler ist

```
gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);
```

(`fovy` $= \theta$ heißt *Field Of View angle in Y-direction*, `aspect` $= \frac{w}{h}$ ist das Breite-zu-Höhe Verhältnis). Einschränkend zur allgemeinen Kegelstumpfbeschreibung ist der Blick immer entlang der mittleren Symmetrieachse gerichtet, d.h. `right = -left` und `top = -bottom`. Dadurch vereinfacht sich die Transformationsmatrix zu

$$\begin{pmatrix} \cot\frac{\theta}{2}\frac{h}{w} & 0 & 0 & 0 \\ 0 & \cot\frac{\theta}{2} & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zFar-zNear} & -\frac{2\,zFar\,zNear}{zFar-zNear} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

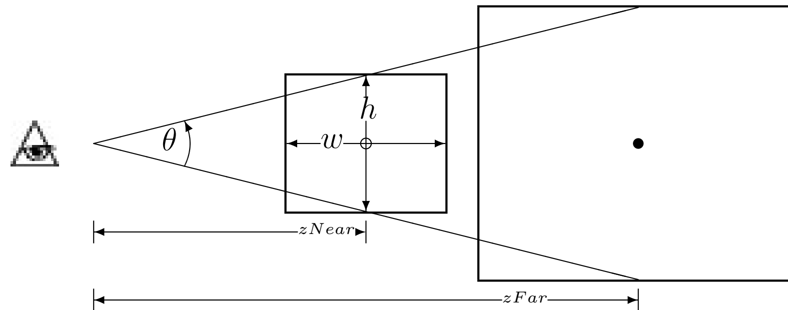


Abbildung 2.7. Zentralperspektivische Projektion.

2.4.4 Belichten des Abzugs

Der *Viewport* bestimmt die Größe und den Ort des endgültigen Bildes (in Fensterkoordinaten).

```
glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

Die *Voreinstellung* des Viewports entspricht den Koordinaten des Fensters. Wählt man den Bereich kleiner oder größer, verändert man die Darstellung innerhalb des Fensters, z.B. bei kleinerem Viewport erscheint das Objekt angeschnitten. Wenn Sie die Größe ihres Fensters verändern (mit der Maus aufziehen), dann sollten Sie nach

```
glutDisplayFunc();
```

```
glutReshapeFunc();
```

mit einem entsprechenden Funktionsaufruf `reshape();` versehen, der die Größe Ihres Viewports kennt. Sie können mittels des Viewports auch Verzerrungen erzeugen, wenn

$$\frac{right_F - left_F}{top_F - bottom_F} = \frac{w_P}{h_P} \neq \frac{w_V}{h_V}.$$

2.5 Matrix Stack in OpenGL

Merke: Alle Transformationen werden über 4×4 Matrizen erzeugt. Diese werden als Vektor der Länge 16 gespeichert. Die einzelnen Einträge finden sich nacheinander in den Spalten der Transformationsmatrix wieder.

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}.$$

Einige der in OpenGL für das Arbeiten mit diesen Matrizen benutzten Aufrufe lauten:

```
glLoadIdentity();

glLoadMatrix*();

glMultMatrix*();
```

Jede der typischen Transformationen wie Drehen, Skalieren und Verschieben erzeugt mit den dabei übergebenen Parametern eine Transformationsmatrix. Diese wird mit der aktuellen Matrix multipliziert. Man kann natürlich auch eine beliebige Matrix vorgeben und über Multiplikation eine eigene Transformation vornehmen. Das Resultat wird zur *current matrix*.

Tipp: Undo-Funktionen sollten auf zuvor gespeicherte Matrizen zurückgreifen, womit a) die Performance heraufgesetzt und b) das Akkumulieren von Fehlern beim häufigen Anwenden von Matrixmultiplikationen oder gar beim Invertieren von Matrizen vermieden werden kann.

Nacheinander ausgeführte Transformationen sind *Linksmultiplikationen* an die Vektoren des 3D-Objekts.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(N);
glMultMatrixf(M);
glMultMatrixf(L);
glBegin(GL_POINTS);
   glVertex3f(v);
glEnd();
```

Als Formel notiert sich die Operation kurz als

$$v' = I(N(M(Lv))) = NMLv$$

wobei das Produkt die *current Modelview-Matrix* ergibt.

Wichtig: *Viewing* (Blickrichtung, siehe 2.4.1) und *Modelling* Transformationen (Szenenarrangement, siehe 2.4.2) erzeugen Multiplikationen mit der *Modelviewmatrix* `glMatrixMode(GL_MODELVIEW)`.

Projektions- (Linseneinstellung, siehe 2.4.3) und *Viewport* Transformationen (Belichten, siehe 2.4.4) erzeugen Multiplikationen mit der Projektionsmatrix `glMatrixMode(GL_PROJECTION)`. Hier entsteht das *Clipping*.

Innerhalb der OpenGL-Maschine ist Speicherplatz für Vielfache beider Matrizen vorgesehen, die sogenannten *Matrix Stacks*. Der *Stack* für die *Modelview* Matrix hat die Länge 32, der *Stack* der *Projection* Matrix sieht nur zwei Matrizen vor. Diese Vorgaben ergeben sich aus der Tatsache, dass man im wesentlichen nur zwei wirklich unterschiedliche Projektionen kennt, die orthographische und die perspektivische, aber natürlich sehr viel mehr Transformationen des Models und der Kameraposition denkbar und auch sinnvoll zu speichern sind.

2.5.1 Arbeiten mit dem Matrixstack

Anstatt selber eine Routine zu schreiben, die die Matrizenverwaltung leistet, kann man auf die Routinen

```
glPushMatrix();
.
glPopMatrix();
```

zurckgreifen. Das *Push* verdoppelt die aktuelle Matrix, und (d)rückt alle im *Matrixstack* befindlichen Matrizen um eine Stelle auf. Das *Pop* löscht die aktuelle Matrix und rückt alle zuvor im Matrixstack befindlichen Matrizen um eine Stelle herab. Ein Paar aus *Push und Pop*, das zwischenzeitlich keine Transformationsänderungen und kein Zeichnen zulässt, hat keine Auswirkungen.

ERRORS: *Matrixstackoverflow* entsteht, wenn zu häufig `glPushMatrix();` gerufen wurde, *Matrixstack is empty*, wenn `glPopMatrix();` alle Matrizen entfernt hat.

*Abhilfe:*Benützen Sie *Push/Pop*-Routinen immer **paarweise**. Vergewissern Sie sich mit

```
glGetBooleanv(GLenum pname, GLBoolean* params);
```

(hier also beispielsweise `glGetBooleanv(GL_MATRIX_MODE, &matrixmode);`), welche Transformationsmatrix gerade beeinflusst wird. Voreingestellt, weil meist benutzt, ist die *Modelview* Matrix.

Der Einsatz von *Push/Pop*-Routinen wird zu folgenden Zwecken benutzt:

- **Umschalten zwischen orthographischer und perspektivischer Projektion.**
- **Ausnützen von Symmetrieeigenschaften der 3D-Objekte beim Zeichnen:** Hier handelt es sich um den Aufbau von *statischen Objekten*, die auch in *Displaylists* abgelegt sein können. Dabei wird das Objekt aus seinen gegeneinander transformierten Teilen aufgebaut - diese Transformationen sollen die Verschiebung des gesamten Objekts *nicht* beeinflussen.

```
push
   rotate
   draw
   rotate
   draw
   .
pop
```

Transformationen in unabhängigen Koordinatensystemen: Hier will man *animierte Bilder* erzeugen, in denen sich ähnliche Objekte unabhängig voneinander bewegen, z.B. ein Sonnensystem, in dem Planeten gegeneinander verschobene, unterschiedlich große Kugeln sind und verschieden geneigte Eigenrotationen sowie die Rotation um die Sonne vollführen (siehe auch "Red Book", p.144, `planet.c`).

- **Gezieltes Hintereinanderschalten von Transformationen.** Hier ein Pseudocode mit OpenGL-Aufrufen:

```
while(key)
   push
      transform
      glGetFloatv(GL_MODELVIEW_MATRIX,myMatrix);
      draw
   pop
then
   glLoadMatrix(myMatrix);
```

2.6 Übungsaufgaben

Aufgabe 2.1 Keyboard–Callback

Registrieren Sie für die Tastatur einen Callback via

`glutKeyboardFunc(void (*func)(unsigned char key, int x, int y));.`

Dieser Callback soll eine Winkelvariable α um $+10.0$ bzw. -10.0 verändern (Keys: R und L). Zeichnen Sie ein Polygon, das sich um den jeweiligen Winkel dreht.

Erweitern Sie Ihr Keyboard–Menü um die Eingabe von `q`*, die Ihr Programm beendet.*

Aufgabe 2.2 Popup-Menü

Entwerfen Sie ein Popup-Menü für die rechte Maustaste. Dieses Menü soll ein Untermenü `Color` *mit drei Einträgen* `red, green, blue` *enthalten, mit dem die Farbe des Polygons ausgewählt werden kann. Zudem enthält das Hauptmenü einen Menüpunkt* `Quit`*, mit dem sich das Programm beenden lässt.*

Aufgabe 2.3 3D–Ansichten

Schreiben Sie ein Programm, das am Ursprung einer 3–D–Modellwelt ein symbolisches Koordinatensystem darstellt. Erstellen Sie eine orthogonale und eine perspektivische Ansicht.

Richten Sie beide Ansichten innerhalb Ihres Programms ein und ermöglichen Sie ein Umschalten zwischen den Perspektiven (via Mouse oder Keys). Platzieren Sie innerhalb Ihrer Modellwelt einen Standardkörper (z.B. einen Würfel).

Aufgabe 2.4 Rotationsmatrizen

Sei u ein Richtungsvektor im $\mathbb{R}^3$, $\phi \in (-\pi, \pi]$ ein Drehwinkel.

1. *Berechnen Sie die 3×3–Drehmatrix $A(u, \phi)$ einer Rotation um u mit dem Winkel ϕ.*
2. *Berechnen Sie eine Zerlegung von A in Elementarrotationen um die Koordinatenachsen:* $A(u, \phi) = R_x(\phi_x) * R_y(\phi_y) * R_z(\phi_z)$.
3. *Geben Sie ein Beispiel für eine Rotation $A(u, \phi)$, für die die Zerlegung in die drei Elementarrotationen nicht eindeutig ist.*
4. *Zeigen Sie: Jede Rotation um die y–Achse kann durch geeignete Rotationen um die x– und die z–Achse ersetzt werden.*
   ***Bem.**: Für die Rotationsgruppe sind also zwei Erzeuger ausreichend.*

Aufgabe 2.5 3D–Anwendungsbeispiel

Schreiben Sie ein Programm, das den Blick auf einen $2 \times 2 \times 2$–Drahtgitter–Würfel im Ursprung einer 3–D–Modellwelt zeigt. Auf Tastendruck `p` *platzieren Sie in diesem Würfel 100 kleine Kugeln zufällig. Die Farbe dieser Kugeln (grün bzw. rot) richtet sich danach, ob der Abstand des Mittelpunkts vom Koordinatenursprung < 1.0 oder ≥ 1.0. ist. Richten Sie eine weitere Taste* `n` *zur Neuplatzierung aller Kugeln ein, die nur aktiv ist, wenn schon Kugeln gesetzt wurden.*

Aufgabe 2.6 Perspektivische Projektion

Die perspektivische Berechnung der 3D Koordinaten geschieht in OpenGL entweder durch Angabe des Winkels, des Seitenverhältnisses und der Tiefeninformation oder mittels der Abmessungen des

darzustellenden Quaders. Diese Angaben werden zu Einträgen der Projektionsmatrix vom Graphiksystem mittels trigonometrischer Formeln umgerechnet. Stellen Sie eine Gleichung für den Winkel auf, und formulieren sie das Ergebnis in Merksätzen (Ein großer Winkel bedeutet ..., ist der Winkel klein, dann ... . Abgabe als Textfile.)

Aufgabe 2.7 Roboterarm

Bauen Sie einen (statischen) Roboterarm als 3D-Drahtgitterobjekt aus Polygonzügen zusammen, indem Sie wiederkehrende Elemente durch Verschieben, Drehen und Skalieren aneinandersetzen. Dieser Arm soll mindesten drei Gelenke und zwei Finger haben. Arbeiten Sie dabei mit dem Matrixstack. (Vorübung für eine spätere Aufgabe.)

Abbildung 2.8. Beispiel eines Industrieroboters

Aufgabe 2.8 Doppelter Viewport

Legen Sie zwei Viewports in ein Fenster. Weisen Sie den Viewports verschiedene Inhalte zu, beispielsweise ein 3D-Objekt in den linken, Text in den rechten Viewport. Experimentieren Sie dabei mit `glOrtho(..);` `glFrustum(..);` *sowie den* Utilities `gluPerspective(..);` `gluLookAt(..);` `gluOrtho2D(..);` *Hinweis: Benutzen Sie* `glScissor(GLint x, GLint y, GLsizei width, GLsizei height);`*, damit sich beim Neuzeichnen durch* `glutReshapeFunc(myReshape);` *der Aufruf von* `glClear(..);` *nur auf den bezeichneten Bereich auswirkt.*

Kapitel 3

Zeichenalgorithmen

3.1 Zeichnen von Linien

Zu den 2D-Rastergraphik-Basics gehören die inkrementelle Algorithmen zum Zeichnen von Linien. Ein *brute force* Algorithmus, der mit der Steigungsform der Geraden $f(x) = mx + B$ arbeitet, würde nach dem Schema ablaufen:

- Berechne die Steigung $\quad m = \Delta y / \Delta x$
- Starte von links $\quad y_i = m\, x_i + B$
- Zeichne den Pixel $\quad (x_i, Round(y_i))$

Dabei meint $Round(y_i) = Floor(0.5 + y_i)$. Der nächste Pixel wird inkrementell ermittelt über

$$y_{i+1} = m\, x_{i+1} + B = m(x_i + \Delta x) + B = y_i + m\Delta x$$

wobei das Inkrement mit $\Delta x = 1$ gerade $m\Delta x = m$ ist. Die Nachteile sind offensichtlich:

- y und m müssen als `REAL` *(fractional binaries)* gespeichert werden.
- Der anschließende Rundungsprozess auf Integer braucht Zeit.

3.1.1 Midpoint Line Algorithmus

Die *Scanline* verläuft von links nach rechts und baut das Bild von oben nach unten auf. Die Idee besteht darin, einen Algorithmus zu entwickeln, der das Raster des Bildschirms und die Technik des

Bildaufbaus ausnutzt. Der *Midpoint Line Algorithmus* wird häufig auch als *Bresenham* Algorithmus bezeichnet, da sich das von Van Aken (1985) entwickelte Verfahren auf seine Formulierung von 1962 zurückführen lässt.

Abbildung 3.1. Jack Bresenham

- Zeichne eine Linie von (x_0, y_0) nach (x_1, y_1)
- (x_p, y_p) sei ein bereits ausgewählter Pixel auf dieser Linie

Wir werden uns auf den Fall einer positiven Steigung $m \in]0, 1[$ beschränken, alle anderen Linien lassen sich aus Symmetrieüberlegungen auf diesen Fall zurückführen.

Die Steigungsform einer Linie

$$y = \frac{dy}{dx} x + B \quad \text{mit} \quad dy = y_1 - y_0,\ dx = x_1 - x_0$$

wird zunächst in die implizite Formulierung umgeschrieben:

$$\boxed{\begin{array}{rcl} F(x,y) & = & dy\,x - dx\,y + B\,dx = 0 \\ & = & a\,x \; + \; b\,y \; + \; c \;\; = 0 \end{array}} \tag{3.0}$$

Damit ist $F(\mathtt{M}) = F(x_p + 1, y_p + 1/2) = d$ der Funktionswert in M und sein Vorzeichen entscheidet über den nächsten Punkt auf der Linie.

Wenn die Entscheidungsvariable $d > 0$ wähle NE
Wenn die Entscheidungsvariable $d \leq 0$ wähle E

Der nächste Schritt im Bildschirmraster richtet sich danach, ob E oder NE ausgewählt wurde. Man berechnet die neue Entscheidungsvariable aus der alten mit

- Wenn E ausgewählt ist

$$\begin{array}{rcccl} d_{new} & = & F(x_p + 2, y_p + 1/2) & = & a(x_p + 2) + b(y_p + 1/2) + c \\ d_{old} & = & F(x_p + 1, y_p + 1/2) & = & a(x_p + 1) + b(y_p + 1/2) + c \end{array}$$

$$\Rightarrow d_{new} = d_{old} + a \quad mit \quad a = \Delta\mathtt{E} = dy$$

- Wenn NE ausgewählt ist

$$\begin{array}{rcccl} d_{new} & = & F(x_p + 2, y_p + 3/2) & = & a(x_p + 2) + b(y_p + 3/2) + c \\ d_{old} & = & F(x_p + 1, y_p + 1/2) & = & a(x_p + 1) + b(y_p + 1/2) + c \end{array}$$

$$\Rightarrow d_{new} = d_{old} + a + b \quad mit \quad a + b = \Delta\mathtt{NE} = dy - dx$$

Zum Initialisieren fehlt noch der Wert einer anfänglichen Entscheidungsvariablen. Sie bestimmt sich aus dem Funktionswert bei (x_0, y_0) und kann, da es nur auf das Vorzeichen ankommt, ganzzahlig gemacht werden.

$$\begin{array}{rcccl} d_{ini}/2 & = & F(x_0 + 1, y_0 + 1/2) & = & a(x_0 + 1) + b(y_0 + 1/2) + c \\ & & & = & F(x_0, y_0) + a + b/2 \\ & & d_{ini} & = & 2\,a + b = 2\,dy - dx \end{array}$$

Somit ergibt sich der folgende `Pseudocode`:

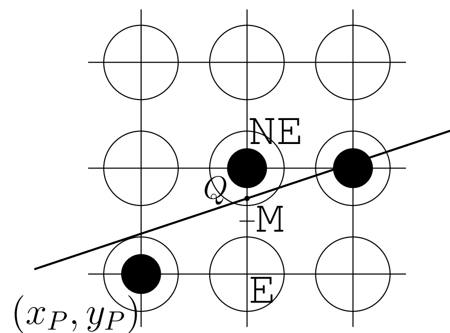


Abbildung 3.2. Der *Midpoint* `M` zwischen `East` und `NorthEast`.

```
void MidpointLine (int x0, int y0, int x1, int y1, int value)
{
  int dx, dy, IncrE, IncrNE, d, x, y;
  dx = x1 - x0;              /* Initialize */
  dy = y1 - y0;
  d  = dy*2 - dx;
  IncrE  = dy*2;
  IncrNE = (dy - dx)*2;
  x = x0;
  y = y0;
  WritePixel(x, y, value);   /* First Pixel */
  while (x < x1){
    if (d <= 0) {           /* Choose E */
       d += IncrE;
       x++;
    } else {                /* Choose NE */
       d += IncrNE;
       x++;
       y++;
    }
    WritePixel(x, y, value); /* Next Pixel */
  }
}
```

3.1.2 Doppelschrittalgorithmus

Der Doppelschrittalgorithmus schaut auf das nächste Paar von Pixeln und kennt deshalb vier mögliche Muster. Die Muster (1) und (4) können nicht auf der gleichen Linie liegen. Deshalb unterscheidet man zunächst zwei Fälle:

1. Fall	Wenn die Steigung $m < 1/2$ wähle aus den Mustern	(1), (2), (3)
2. Fall	Wenn die Steigung $m \geq 1/2$ wähle aus den Mustern	(2), (3), (4)

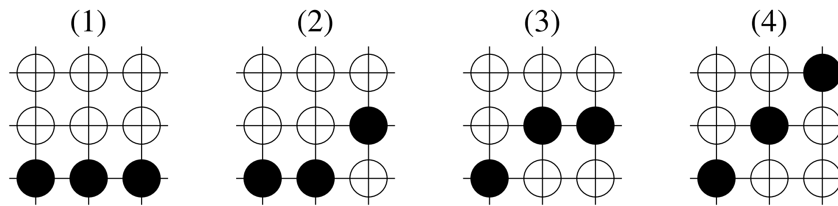


Abbildung 3.3. Die vier Grundmuster beim Doppelschrittalgorithmus.

Für das Doppelschrittverfahren und den 1. Fall ergibt sich der folgende `Pseudocode`:

```
void DoubleStep (int x0, int y0, int x1, int y1)
{
  int currentx, Incr1, Incr2, cond, dx, dy, d;
  dx = x1 - x0;            /* Initialize */
  dy = y1 - y0;
  currentx = x0;
  Incr1 = 4*dy;
  Incr2 = 4*dy - 2*dx;
  cond = 2*dy;
  d = 4*dy - dx;
  while (currentx < x1){
    if (d < 0) {           /* Choose Pattern (1) */
       DrawPixels(Pattern1,currentx);
       d += Incr1;
    }
    else {
       if (d < cond)       /* Choose Pattern (2) */
          DrawPixels(Pattern2,currentx);
       else                /* Choose Pattern (3) */
          DrawPixels(Pattern3,currentx);
    }
    d += Incr2;
  }
  currentx += 2;
}
```

Bemerkung 3.1 *Wenn man mit* Alphachannel *arbeitet, also über transparente Farben verfügt, kann man den Doppelschrittalgorithmus so modifizieren, dass man nicht zwischen den Mustern (2) und (3) unterscheidet, sondern beide mit halber Intensität zeichnet. Dadurch erhält man ein* Antialiasing *geschenkt.*

3.1.3 Unabhängigkeit der Laufrichtung

Wenn eine Linie direkt durch einen Mittelpunkt verläuft und damit $d = 0$ ist, gilt die folgende Konvention:

Wähle aufsteigend (von links nach rechts) `E`
Wähle absteigend (von rechts nach links) `SW`

Somit wird garantiert, dass bei aufsteigend oder absteigend durchlaufener *Scanline* immer die gleiche Linie gezeichnet wird. Das einzige Problem, dass sich ergibt, bezieht sich auf gepunktete oder gestrichelte Linien. Schreibmasken beginnen immer links unten (siehe dazu auch Aufgabe 3.2).

3.1.4 Schnittpunkte mit Rechtecken

Wenn die Linie durch ein Rechteck begrenzt wird, was immer dann der Fall ist, wenn sie an den Rand des *Viewports* oder des Fensters kommt, soll sie möglichst natürlich aufhören und nicht etwa gekrümmt oder abgehackt erscheinen.

Dazu betrachten wir zunächst die linke Kante eines Rechtecks. Für eine Situation, wie in Abbildung 3.4 dargestellt, muß in die Initialisierung bereits eingehen, dass der durch Rundung gewählte Pixel oberhalb des tatsächlichen Schnittpunkts liegt. Der *Midpoint* Algorithmus muß mit `M` zwischen `E` und `NE` starten.

Für die untere Kante gilt, dass der tatsächliche Schnittpunkt der Linie mit y_{min} nur die schwarzen Pixel zeichnen würde, der den Eindruck einer abgebrochenen Linie hinterlässt. Als Abhilfe erzeugt man einen Schnittpunkt mit $y_{min} - 1/2$, der auch noch die rot dargestellten Pixel zeichnet und dem Muster auf ganzer Linie entspricht.

Hierzu eine historische Anmerkung: Vom *User* wurde früher verlangt, dass er *Viewports* entsprechend von beispielsweise -0.5 bis 255.5 angibt, um die korrekte Darstellung der Linie zu garantieren.

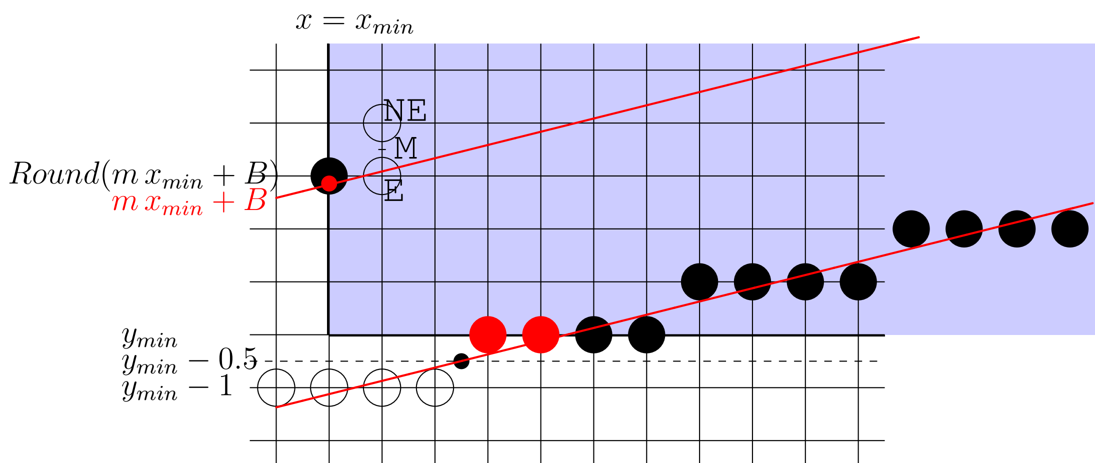


Abbildung 3.4. Handhabe bei der linken und unteren Kante von Fenstern (angeschnittene Rechtecke).

3.1.5 Intensitätsschwankungen

Linien unterschiedlicher Steigung variieren stark in ihrer Intensität, da eine konstante Anzahl von Pixeln jeweils Linien von einer Längeneinheit bis zu maximal $\sqrt{2}$ Längeneinheiten darstellen kann. Das wird besonders deutlich, wenn man die Linienstärke erhöht und dabei die Anzahl der Pixel in x-Richtung meint. Einzig für Linien, die in x-Richtung verlaufen, wird die y-Richtung für die Linienstärke genommen.

Auf einem *Bilevel-Display* gibt es keine Abhilfe, aber auf anderen Schirmen kann man die Intensität einer Linie als Funktion der Steigung auffassen, also $I = I(m)$.

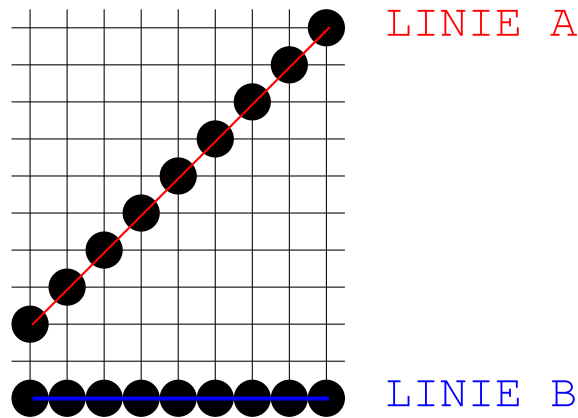


Abbildung 3.5. Variierende Intensitäten bei unterschiedlicher Steigung.

3.2 Zeichnen von Kreisen

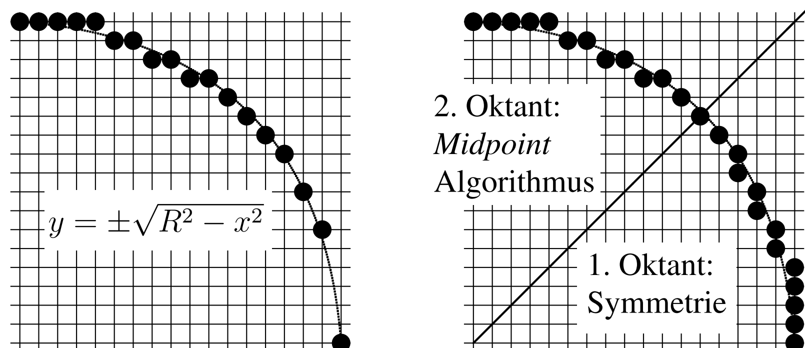


Abbildung 3.6. *Brute Force* Algorithmus und *Midpoint* Algorithmus mit Symmetrien.

Beim Zeichnen von Kreisen sei nochmals auf die Symmetrieeigenschaften der Rasterschirme hingewiesen, wobei nach Konvention hier nicht der 1. Oktanten (wie bei den Steigungen der Linien) sondern der 2. Oktanten berechnet wird.

3.2.1 Symmetrieeigenschaften

```
void CirclePoint (int x, int y, int value)
{
  WritePixel( x, y, value);
  WritePixel( y, x, value);
  WritePixel( y,-x, value);
  WritePixel( x,-y, value);
  WritePixel(-x,-y, value);
  WritePixel(-y,-x, value);
  WritePixel(-y, x, value);
  WritePixel(-x, y, value);
}
```

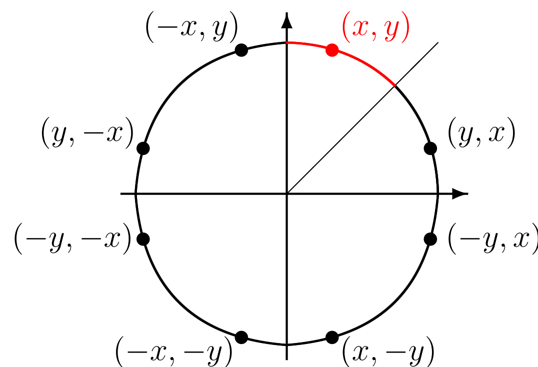


Abbildung 3.7. Symmetriebetrachtungen beim 2D-Kreis auf quadratischem Gitter.

3.2.2 Midpoint Circle Algorithmus

Dieser ebenfalls auf Bresenham zurückgehende Algorithmus aus dem Jahr 1977 wurde 1983 sogar patentiert (Nr. 4 371 933). Wieder wird die implizite Formulierung gewählt und eine Entscheidungsvariable betrachtet.

$$F(x, y) = x^2 + y^2 - R^2 = 0$$

Die Konvention ist hier, den Kreis beginnend bei zwölf Uhr und mit dem Uhrzeigersinn bis ein Uhr dreißig zu berechnen. Daher wird der x-Wert erhöht, der y-Wert vermindert und die Bezeichnungen der Himmelsrichtungspixel lautet `East` und `SouthEast`. $F(\mathtt{M}) = F(x_p + 1, y_p - 1/2) = d$ der Funktionswert in `M` und sein Vorzeichen entscheidet über den nächsten Punkt auf der Linie.

Wenn die Entscheidungsvariable $d \geq 0$ wähle SE
Wenn die Entscheidungsvariable $d < 0$ wähle E

$$d_{old} = F(x_p + 1, y_p - 1/2) = (x_p + 1)^2 + (y_p - 1/2)^2 - R^2$$

- Wenn $d_{old} < 0$ und somit E ausgewählt ist

$$\begin{aligned} d_{new} &= F(x_p + 2, y_p - 1/2) &= (x_p + 2)^2 + (y_p - 1/2)^2 - R^2 \\ &= d_{old} + 2\,x_p + 3 &= d_{old} + \Delta\mathtt{E} \end{aligned}$$

- Wenn $d_{old} \geq 0$ und somit SE ausgewählt ist

$$\begin{aligned} d_{new} &= F(x_p + 2, y_p - 3/2) &= (x_p + 2)^2 + (y_p - 3/2)^2 - R^2 \\ &= d_{old} + 2\,x_p - 2\,y_p + 5 &= d_{old} + \Delta\mathtt{SE} \end{aligned}$$

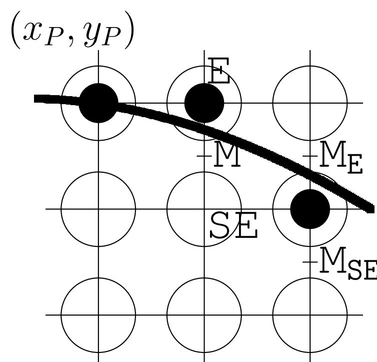


Abbildung 3.8. Der *Midpoint* M zwischen East und SouthEast.

Für die Initialisierung von d berechnet man

$$\begin{aligned} d = F(x_0 + 1, y_0 - 1/2) &= F(1, R - 1/2) &= 1 + (R^2 - R + 1/4) - R^2 \\ & &= 5/4 - R \end{aligned}$$

Damit erhält man für einen Kreis im Ursprung den folgenden Pseudocode:

```
void MidpointCircle(int radius int value)
{
  int x, y;
  float d;
  x = 0;                       /* Initialize */
  y = radius;
  d = 5.0/4.0 - radius;
  CirclePoints(x, y, value);   /* First Pixel */
  while (y > x){
    if (d < 0) {               /* Choose E */
       d += x*2.0 + 3;
       x++;
    } else {                   /* Choose SE */
       d += (x - y)*2.0 + 5;
       x++;
       y--;
    }
    CirclePoints(x, y, value); /* Next Pixel */
  }
}
```

Vergleicht man den *Midpoint Line* mit dem *Midpoint Circle* Algorithmus, stellt sich der *Circle* Algorithmus zunächst nicht als *Integer* Algorithmus dar, was sich aber leicht ändern läßt (siehe Aufgabe 3.1).

	Midpoint Line	*Midpoint Circle*
Offset	$\Delta\mathtt{E} = a = dy \equiv$ const $\Delta\mathtt{NE} = a + b = dy - dx \equiv$ const	$\Delta\mathtt{E} = 2\,x_p + 3$ $\Delta\mathtt{SE} = 2\,x_p - 2\,y_p + 5$
Inkremente	konstant	Lineare Funktionen des Auswertungspunktes
Operationen	Integer	Floating point

Tabelle 3.1. Vergleich der *Midpoint* Algorithmen.

Eine weitere Verbesserung kann vorgenommen werden, wenn man die Inkremente $\Delta\mathtt{E}$ und $\Delta\mathtt{SE}$ zunächst mit $\Delta\mathtt{E} = 3$ und $\Delta\mathtt{SE} = 5 - R * 2$ initialisiert und dann, abhängig davon ob E oder SE ausgewählt ist, beide um zwei oder $\Delta\mathtt{SE}$ um 4 erhöht. Bei der Kreisgleichung handelt es sich um eine quadratische Gleichung, deshalb sind nicht die ersten sondern die zweiten Ableitungen konstant. Also muß die Entscheidungsvariable d um ein ständig wachsendes $\Delta(\mathtt{S})\mathtt{E}$ vergrößert werden.

(x_p, y_p)	Wähle E$\rightarrow (x_p+1, y_p)$	
$\Delta\text{E}_{old} = 2\,x_p + 3$ $\Delta\text{SE}_{old} = 2\,x_p - 2\,y_p + 5$	$\Delta\text{E}_{new} = 2\,(x_p+1) + 3$ $\Delta\text{SE}_{new} = 2\,(x_p+1) - 2\,y_p + 5$	$\Delta\text{E}_{new} - \Delta\text{E}_{old} = 2$ $\Delta\text{SE}_{new} - \Delta\text{SE}_{old} = 2$

(x_p, y_p)	Wähle SE $\rightarrow (x_p+1, y_p-1)$	
$\Delta\text{E}_{old} = 2\,x_p + 3$ $\Delta\text{SE}_{old} = 2\,x_p - 2\,y_p + 5$	$\Delta\text{E}_{new} = 2\,(x_p+1) + 3$ $\Delta\text{SE}_{new} = 2\,(x_p+1) - 2\,(y_p-1) + 5$	$\Delta\text{E}_{new} - \Delta\text{E}_{old} = 2$ $\Delta\text{SE}_{new} - \Delta\text{SE}_{old} = 4$

Der `Pseudocode` muß jetzt wie folgt abgeändert werden.

```
if (d < 0) {        /* Choose E */
    d += deltaE;
    deltaE += 2;
    deltaSE += 2;
    x++;
} else {            /* Choose SE */
    d += deltaE;
    deltaE += 2;
    deltaSE += 4;
    x++;
    y--;
}
```

3.3 Antialiasing von Linien und Flächen

Der Begriff *Aliasing* wird verwendet, weil ein Pixel entweder zum einen oder zum anderen Objekt gehört, also seine Zugehörigkeit mit einem Objektnamen belegt werden kann. Diese ALLES-ODER-NICHTS Entscheidung führt zu dem Phänomen der Sägezähne (= *jaggies)*, das man auch *Staircasing* nennt. Bei Linien ist dieser Effekt besonders auffällig, wobei ganz gering steigende Linien (oder nahezu senkrechte Linien), die nur an wenigen Stellen um einen Pixel springen, besonders unangenehm auffallen.

Eine mögliche Abhilfe wäre das Erhöhen der Bildschirmauflösung. Das Halbieren der Pixelgröße benötigt aber bereits einen vierfachen Bildschirmspeicher, abgesehen davon, dass man Bildschirmpixel nicht beliebig klein machen kann.

Allerdings kann man die Intensität eines Pixels variieren, ihm also anteilig die Intensitäten zuordnen, die dem Pixelanteil am jeweiligen Objekt entspricht. Ein solches Vorgehen wird als *Antialiasing* bezeichnet. Die Flächenanteile werden abgetastet und im einfachsten Fall ungewichtet und linear kombiniert.

Da Flächen und Linien mit unterschiedlichen Algorithmen erzeugt werden, beansprucht eine gefüllte Fläche unter Umständen andere Pixel am Rand, als das Drahtgittermodell derselben Fläche. Um Schwierigkeiten beim Darstellen von Flächen und Linien zu vermeiden, beschränkt man das *Antialiasing* meist nur auf die Linien und gleicht mit der Hintergrundfarbe ab. An den Rändern von Flächen wird auf das Glätten verzichtet, da es hier meist nicht besonders störend wirkt. So kann man auf komplizierte Sortieralgorithmen verzichten, denn beim Drehen einer Szene können unterschiedlich farbige Objekte voreinander geschoben werden und damit müssten die jeweiligen Objektfarben gespeichert werden.

Genügt dieses Flächenabtasten nicht den Ansprüchen, so kann man wie beim Doppelschrittalgorithmus den *Alphachannel*, also das transparente Überblenden geschickt ausnutzen. Für dieses sogenante Szenen-Antialiasing arbeitet man mit dem *Accumulationbuffer* (siehe im Kapitel über Bufferkonzepte).

3.3.1 Ungewichtete Flächenabtastung

Eine ideale Linie hat keine Breite. Die tatsächliche Linienstärke auf einem Bildschirm ist aber immer größer oder gleich einem Pixel. Dabei nimmt man an, dass die Pixel disjunkte Quadrate sind, und errechnet daraus die Flächenanteile.

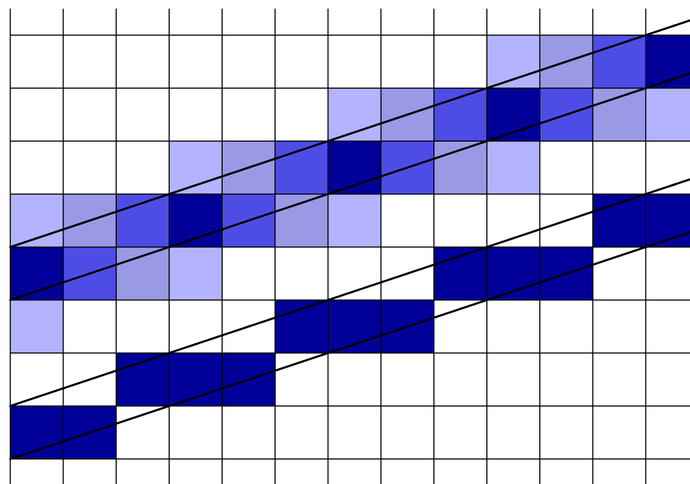


Abbildung 3.9. Linie mit *Antialiasing* (oben) und ohne *Antialiasing* (unten).

1. Die Intensität des Pixels ist eine lineare Funktion der am Objekt beteiligten Fläche.
2. Fern- oder Strahlwirkung wird ausgeschlossen (ETWAS-ODER-NICHTS Entscheidung).
3. Der Abstand der angeschnittenen Fläche zum Mittelpunkt des Pixels geht nicht ein (Steigungsunabhängigkeit).

3.3.2 Gewichtete Flächenabtastung

Im Gegensatz zur ungewichteten Flächenabtastung werden entsprechend der obigen Auflistung die folgenden Verbesserungen vorgenommen:

1. Überlapp wird zugelassen.
2. Statt *Boxweighted* = *unweighted* wird mit einem Konus oder einer Gaussschen Glockenfunktion gewichtet.
3. Geringer Abstand der beteiligten Fläche zum Zentrum des Pixels hat stärkeren Einfluss.

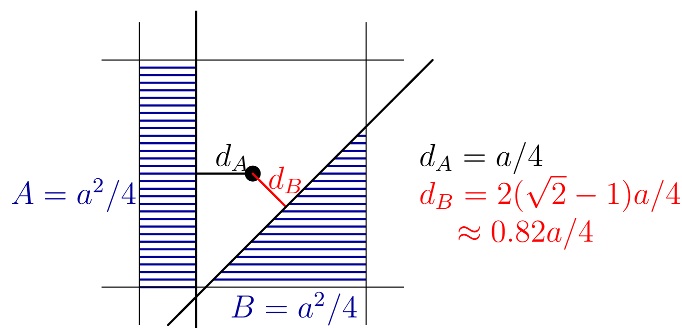


Abbildung 3.10. Gleichgroße Flächen mit unterschiedlichem Abstand zum Mittelpunkt des Pixels.

3.4 Das Zeichnen von Polygonen

Die auf den Schirm projizierten Polygone werden entlang der sogenannten *Scanline* gezeichnet. Der Algorithmus gliedert sich dabei in die folgenden Schritte:

1. Finde alle Schnittpunkte der *Scanline* mit den Kanten des Polygons.
2. Sortiere die Schnitte nach aufsteigenden x-Werten.
3. Fülle mit Farbwerten nach der Paritätsregel.

Die *Paritätsregel* meint dabei, dass bei gerader Parität KEINE Pixel gezeichnet werden, bei ungerader Parität dagegen werden sie gezeichnet. Man beginnt mit gerader Parität. Jede Schnittstelle mit einer Kante des Polygons invertiert die Parität.

Definition 3.1 *In der Geometrie nennt man ein Polygon P in der Ebene* monoton *in Bezug auf eine Gerade L, wenn jede zu L orthogonale Linie das Polygon P höchstens zweimal schneidet.*

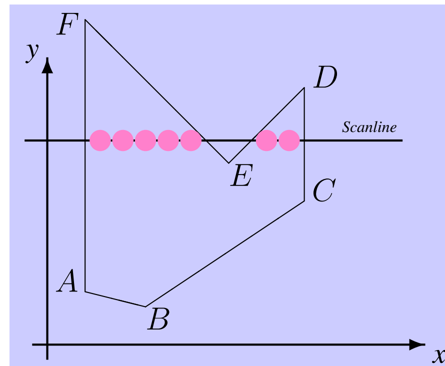


Abbildung 3.11. Entlang der Scanline wird mit der Paritätsregel das Polygon gefüllt.

Aus vielen praktischen Gründen erweitert man diese Definition dahin, dass die Fälle von Polygonen, bei denen manche der Kanten von P orthogonal zu L liegen und damit unendlich viele Schnittpunkte existieren, auch zu den monotonen Polygonen bezüglich L gerechnet werden. Zum Beispiel ist das Polygon aus Abbilding 3.11 monoton bezüglich der x-Achse, aber nicht monoton bezüglich der y-Achse. Konvexe Polygone sind bezüglich jeder Geraden, also auch bezüglich beider Koordinatenachsen monoton. Da die Polygone durch Drehung in immer wieder veränderter Projektion auf dem Schirm erscheinen, lassen sie sich einfacher und schneller darstellen, wenn man davon ausgeht, dass sie konvex sind.

In der Rastergraphik, also nach der Projektion der 3D Welt auf Schirmkoordinaten sind alle Polygone zunächst planar und aus Dreiecken zusammengesetzt. Ein monotones Polygon kann einfach und in linearer Zeit trianguliert werden. Konvexe Polygone sind auch nach Drehung noch monoton zur y-Achse, so dass die Triangulierung durch unterschiedliche, hardwareabhängige Algorithmen immer ähnlich aussieht. In der *OpenGL* Implementierung funktioniert daher der `GL_POLYGON` Modus zum Zeichnen nur für konvexe Polygone in korrekter Weise.

3.5 OpenGL Implementierung

Da man sich heute nicht selbst um die Implementierung von Linien kümmern muss, ist es vielmehr hilfreich, die Bibliotheksaufrufe in Standardimplementierungen zu kennen. Die Optionen, die der Nutzer hier hat, lassen sich aber besser verstehen, wenn man die Algorithmen dahinter kennt.

3.5.1 Gestrichelte Linien

Um gestrichelte Linien zu zeichnen, kann man ein Muster angeben, das mit einem Faktor skalierbar ist. Grundsätzlich werden Linien nicht gestrichelt, daher muss in *OpenGL* diese Eigenschaft von Linien eingeschaltet werden und ist dann solange mit dem angegebenen Muster wirksam, bis entweder

das Muster geändert oder die Strichelung abgeschaltet wird.

```
glLineStippel(GLint factor, GLushort pattern);
glEnable(GL_LINE_STIPPEL);
```

Der Faktor variiert sinnvoll zwischen 1 und 256. Das Muster *(pattern)* ist eine 16 bit lange Serie von Nullen und Einsen, die das An- und Ausschalten von hintereinander gezeichneten Pixeln bedeutet. Ein Beispiel: die Hexadezimalzahl 0x3F07 (als Dezimalzahl 16135) hat die Binärdarstellung 0011111100000111. Diese Zahl wird von rechts ausgelesen und erzeugt das Muster durch 3 mal an-, 5 mal aus-, 6 mal an- und 2 mal ausgeschaltete Pixel. Ein Faktor zwei verdoppelt die Anzahl zu 6 mal an, 10 mal aus, 12 mal an und 4 mal aus.

3.5.2 Antialiasing

Im Color-Index-Mode, also beim Arbeiten mit indizierten Farben wie sie beispielsweise für GIF-Bilder (Graphics Interchange Format) verwendet werden, muss man beim Antialiasing selbst für entsprechende Farbrampen sorgen, die beim Farbabgleich mit der Hintergrundfarbe an und abgefahren werden. Dazu initialisiert man das Graphikfenster mit `GLUT_INDEX` statt mit `GLUT_RGB`. Eine solche Rampe wird mit 16 fortlaufenden Indizes als Lookup-Table gebraucht. In einem zweiten Schritt gibt man den Startindex für die Rampe zur Linienfarbe an und schaltet Antialising ein.

```
glIndexi(RAMPSTART);
glEnable(GL_LINE_SMOOTH);
```

Einfacher ist das Antialiasing im RGBA-Mode zu implementieren. Hier überlässt man dem Alphakanal die Farben von Linien entsprechend ihrer Steigung an den Rändern in ihrer Intensität zu mindern. Hier besteht der erste Schritt im Einschalten des Blendings.

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Man kann auf lokale Soft- und Hardware-Ausstattung mit den Aufrufen

```
glHint(GL_POINT_SMOOTH, GL_FASTEST);
glHint(GL_POINT_SMOOTH, GL_NICEST);
```

eingehen, die entweder die schönste oder die schnellste Methode des Glättens ansteuern. Wieder muss man im zweiten Schritt das Antialiasing grundsätzlich einschalten, und zwar getrennt für Punkte, Linien oder Polygone, da hierbei ganz unterschiedliche Algorithmen zum Zeichnen angesteuert werden.

```
glEnable(GL_POINT_SMOOTH);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);
```

3.5.3 Linienbreite

Bei eingeschaltetem Antialiasing ist es durchaus sinnvoll, die Linienbreite als Float zu vereinbaren. Für Linien mit (betragsmäßigen) Steigungen $|m| < 1.0$ wird die Breite bezüglich der y-Richtung gemessen, sonst richtet sich die Breite an der x-Achse aus. Schön wäre eine Linienbreite, die senkrecht zur Linie berechnet wird, aber dafür ist der Rechenaufwand viel zu groß.

```
glLineWidth(GLfloat width);
```

Dabei ist die Breite echt größer Null zu wählen, der Defaultwert ist 1.0.

Je nach Implementierung sind auch breitere Linien geglättet darstellbar. Garantiert wird aber nur eine geglättete Breite von Eins. Unterstützte Breite und Granularität des mit dem Raster korrespondierenden Fragments können abgefragt werden.

```
glGetFloatv(GL_ALIASED_LINE_WIDTH_RANGE);
glGetFloatv(GL_SMOOTH_LINE_WIDTH_GRANULARITY);
```

3.5.4 Zeichnen mit Vertices

Um Objekte im 3D-Raum zu zeichnen, arbeitet man mit Punkten, die zu Linien oder Dreiecken verbunden werden können. Volumina (Polyeder) werden daher über ihre Oberfläche aus Dreiecken (Polygonen) erzeugt. Die einzelnen Flächen haben immer eine Orientierung, die über ihre äußere Normale definiert wird. Diese Normale ist ein Vektor, der sich über ein Kreuzprodukt aus den Eckpunkten eines Dreiecks errechnet. Daher ist die Reihenfolge, in der die Eckpunkte aufgeführt werden, für die Definition von Außen und Innen entscheidend. Mathematisch positiv, also entgegen dem Uhrzeigersinn durchlaufene Dreiecke zeigen ihre Vorderseite.

```
glPolygonMode(GLenum face, Glenum mode);
```

Die Richtung der äußeren Normale beeinflusst vor allem die Lichtmodelle. Zum schnelleren Zeichnen kann es allerdings auch sinnvoll sein, Dreiecke gar nicht zu zeichnen, wenn sie von hinten dargestellt werden müssten *(Backface-Culling)*. Das wir u.a. mit `glCullFace(GLenum face)` erreicht. In beiden Aufrufen kann *face* entweder `GL_FRONT`, `GL_BACK` oder `GL_FRONT_AND_BACK` sein, letzteres ist der Default-Wert.

Man kann sich seine Objekte zwecks Debugging nacheinander als Punkte, Linien und Flächen anzeigen lassen. Dazu wechselt man den Polygonmodus *mode* von `GL_POINT` zu `GL_LINE` oder `GL_FILL`, letzteres ist die Default-Einstellung.

Die Punkte zum Erzeugen von Objekten werden nun zwischen

```
glBegin(GLenum mode);
...
glEnd();
```

eingefügt. Dabei ist der Zeichenmodus *mode* jetzt eine der folgenden *Defines*: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS` oder `GL_QUAD_STRIP`.

Die folgende Tabelle zeigt, wie bei einer Anzahl von N Punkten (# N) für die unterschiedlichen Zeichenmodi verfahren wird. Sollte N nicht ganzzahlig durch 2, 3 oder 4 teilbar sein, werden überzählige Punkte ignoriert.

mode	bezogen auf # N	Art der Darstellung	Einzelteile
GL_POINTS	# N	Punkte	
GL_LINES	# N/2	Linien	
GL_LINE_STRIP	1	offener Linienzug	# N -1 Linien
GL_LINE_LOOP	1	geschlossener Linienzug	# N Linien
GL_TRIANGLES	# N/3	Dreiecke	
GL_TRIANGLE_STRIP	1	Streifen	# N - 2 Dreiecke
GL_TRIANGLE_FAN	1	Fächer	# N - 2 Dreiecke
GL_QUADS	# N/4	Vierecke	
GL_QUAD_STRIP	1	Streifen	# N/2 - 1 Vierecke
GL_POLYGON	1	konvexes Polygon	beliebige Dreiecke

Tabelle 3.2. Spezifikation der Zeichenmodi.

Im Modus `GL_LINES` wird jeweils eine Linie zwischen dem ersten und zweiten, dann dem dritten und vierten, also dem Punkt $2n - 1$ und dem Punkt $2n$ gezeichnet. Jeder Punkt mit ungeradem Index stellt somit den Beginn, jeder Punkt mit geradem Index das Ende einer Linie dar. Ähnlich verfahren `GL_TRIANGLES` und `GL_QUADS`, bei denen jeweils drei (vier) Punkte zu einem Dreieck (Viereck) verbunden werden.

Erst `GL_LINE_STRIP` setzt beim Zeichnen nicht mehr ab und macht aus dem Endpunkt einer Linie den Startpunkt einer neuen Linie. Ähnliches gilt für den `GL_TRIANGLE_STRIP`, der für jeden neu hinzukommenden Punkt in der Liste einen alten vergisst. So wird ein geschlossener Streifen von Dreiecken gezeichnet. Damit der Streifen aber nicht ständig abwechselnd seine Vorder- und Rückseite zeigt, weiß der Algorithmus, dass bei ungeradem n die Punkte n, $n + 1$ und $n + 2$ zu einem Dreieck n verbunden werden müssen, während bei geradem n die Punkte $n+1$, n und $n+2$ das Dreieck n ergeben. Beim `GL_TRIANGLE_FAN` wird der erste Punkt zum Zentrum des Fächers. Das Dreieck n besteht also aus den Punkten 1, $n + 1$ und $n + 2$. Der Drehsinn ist damit immer gleich. Bei `GL_QUAD_STRIP` wird mit einem Punktepaar gestartet und für jedes neue Paar ein Viereck gezeichnet.

Im Modus `GL_POLYGON` wird schließlich ein konvexes Polygon aus der Punkteliste gezeichnet, was bei nichtkonvexen Punktwolken, die sich nicht in einer Ebene befinden aus unterschiedlichen Perspektiven betrachtet beliebig falsch aussehen kann. Hier wird nämlich der Graphikhardware überlassen,

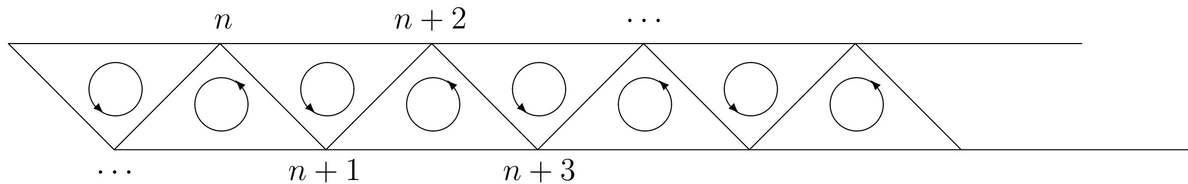


Abbildung 3.12. Der Trianglestrip.

welche Dreiecke zwischen den Punkten zu zeichnen sind. Mit verschiedener Projektion kann daher mal das eine, mal das andere Dreieck sinnvoller (weil großflächiger) erscheinen. Wenn man also nicht völlig sicher ein konvexes, ebenes Polygon vor sich hat, sollte man immer in Streifen oder Fächern zeichnen.

3.6 Übungsaufgaben

Aufgabe 3.1 Bresenham–Algorithmus

Schreiben Sie einen reinen Integer *Algorithmus, der Ihnen die Punkte für einen Kreis mit Integerradius liefert, und zeichnen Sie diese Punkte.*

Zeichnen sie daneben einen Kreis mithilfe von

```
gluDisk(GLUquadric* quad,
        GLdouble inner,
        GLdouble outer,
        GLint slices,
        GLint loops);
```

im Polygonmodus `GL_LINE` *sowie einen zweiten Kreis unter Verwendung eines simplen Algorithmus, der das Intervall* $[0, 2\pi]$ *gleichmäßig unterteilt und jeweils die Position der Kreispunkte in Polarkoordinaten ermittelt.*

Lassen Sie sich die CPU-Zeit der jeweilen Zeichenoperationen an den Schirm ausgeben. Um welchen Faktor sind die ersten beiden Algorithmen schneller?

Aufgabe 3.2 Steigung und Strichelung von Linien

Zeichnen Sie Linien in verschiedensten Steigungen und Strichelungen. Nutzen Sie alle drei Dimensionen. Zeichnen Sie seitlich versetzt diese gleichen Linien mit eingeschaltetem Antialiasing. Benutzen Sie Fog, um Tiefenwirkung zu erzielen.

Drehen Sie Ihre Linien mittels Popup-menü und Tastatursteuerung, wobei Sie die Shortcuts durch Angabe der entsprechenden Taste in ihrem Menüeintrag verdeutlichen.

Aufgabe 3.3 Line-Stitching

Mit Stitching *bezeichnet man u.a. den Effekt, der aus den unterschiedlichen Algorithmen beim Zeichnen von Linien und Polygonen entsteht.*

Zeichnen Sie mit eingeschaltetem Depth Buffer *einen (großen) Raum mit vier Wänden und einer Bodenplatte. Machen Sie die Bodenplatte größer als den Raum, sodass sie an mindestens einer Seite hervorsteht. Verwenden Sie beliebige Farben (außer Schwarz) zum Füllen ihrer Polygone und zeichnen Sie für jedes Polygon die Außenkante in schwarzer Farbe. Variieren Sie die Linienbreite zwischen 1 und 9 durch ein Menü oder die Tastatur. Drehen Sie* entweder *auf Tastendruck die gesamte Szene* oder *ermöglichen Sie eine Kamerabewegung innerhalb der Szene. Beim Drehen oder Bewegen treten nun unschöne Effekte in den Ecken und Kanten der Polygone auf. Verbessern Sie die Darstellung und ermöglichen Sie ein Umschalten zwischen der unschönen und der schönen Darstellung über das Menü.*

Kapitel 4

Bufferkonzepte

Buffer bedeutet Zwischenspeicher, Puffer, auch im Sinne von Dämpfer. In der Computergraphik definiert man die gleichmäßig für alle Pixel gespeicherten Daten als *Buffer* und unterscheidet dabei einzelne Buffer anhand der Algorithmen, die diesen Speicherplatz benötigen.

Definition 4.1 *Die gleichmäßig für alle Pixel gespeicherten Daten nennt man* Buffer*, der* Framebuffer *umfasst alle Buffer.*

Die *Buffer* sind grundsätzlich manipulierbar, treten jedoch an unterschiedlichen Stellen der Graphik-Pipeline in Erscheinung. Der einzig sichtbare *Buffer* ist der *Colorbuffer*, wobei dieser sich je nach *Displaymodus* (und *Hardware-Konfiguration*) beim *Double Buffering* und bei Stereovision in die Buffer für *Front (Front Left* und *Front Right)* sowie *Back (Back Left* und *Back Right)* unterteilt. Der *Depth Buffer* stellt eine elegante Methode zur Darstellung und Verdeckung räumlich hintereinander angeordneter Objekte dar. Mit dem *Stencil Buffer* werden Masken, Schablonen definiert. Je nach Wert in der Schablone werden weitere Operationen zugelassen. Der *Accumulation Buffer* erlaubt das Anhäufen von vielen einzelnen *Frames*, die dann mit einer gewünschten Gewichtung an den sichtbaren *Colorbuffer* weitergegeben werden.

4.1 Double (und Stereo) Buffering

Der Aufbau eines Bildes braucht Zeit, die immer noch zu lange dauert, als dass unser Gehirn sie nicht mehr wahrnehmen würde. Das Darstellen eines fertigen Bildes auf dem Schirm liegt unterhalb der Wahrnehmungsschwelle. Würde man mit einem einzigen Farbspeicher arbeiten, müsste man beim Zeichnen eines Bildes zuschauen. Diese häufig sehr kurze Zeitspanne stört bei einem Standbild überhaupt nicht, sondern kann sogar sehr interessante Aufschlüsse über das Nacheinander einzelner Bildelemente geben (falls die Maschine langsam genug ist). Bei jeder Art von Animation ist aber unbedingt ein *Double Buffer* nötig. Ohne *Double Buffer* zeigt sich ein heftiges Flackern des Bildes bei

der geringsten Transformation. An diesem Flackern erkennt man jederzeit einen Fehler beim Belegen von *Front-* und *Backbuffer*. Möglicherweise besteht die Ursache aber darin, dass das System nur über einen einzigen *Colorbuffer (= Front Left)* verfügt. Bei einem stereofähigen Graphiksystem, müssen sogar vier Buffer korrekt angesprochen werden, da für das linke sowie das rechte Auge unterschiedliche Blickrichtungen und damit unterschiedliche Bilder berechnet und animiert werden müssen.

Bisher wude beim Initialisieren eines Fensters

```
glutInitDisplayMode(...|GLUT_SINGLE|...);
```

aufgerufen. Dieser Aufruf muss von `GLUT_SINGLE` zu `GLUT_DOUBLE` (bzw. zu `GLUT_STEREO`) geändert werden, damit *Double Buffering* (bzw. *Stereo Buffering*) aktiviert wird. Mit `glGetBooleanv();` kann abgefragt werden, ob `GL_DOUBLEBUFFER` und `GL_STEREO` vom System unterstützt werden. Im System müssen dazu zwei (bzw. vier) Kopien des *Colorbuffers* vorhanden sein, die die Bezeichnungen *Front (Front Left und Front Right) Buffer* sowie *Back (Back Left und Back Right) Buffer* tragen. Jedes System stellt zumindest einen *Colorbuffer (= Front Left)* zur Verfügung. Im *Single Buffer Modus* kann man beim Zeichnen zusehen (je nach Leistung und Anforderung).

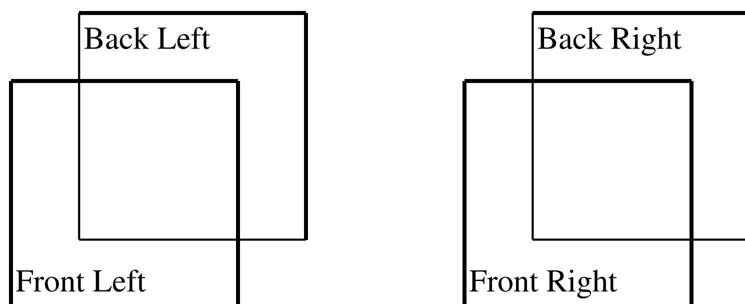


Abbildung 4.1. ***Front Left, Front Right Buffer*** **sowie** ***Back Left, Back Right Buffer*.**

24 bis 25 Bilder *(Frames)* pro Sekunde reichen für die Wahrnehmung einer fließenden Bewegung. Nachdem das Bild im *Single Buffer Modus* gezeichnet wurde, muss es gleich wieder gelöscht werden, um es an leicht veränderter Position neu zu zeichnen. Der visuelle Eindruck ist eine flackernde, ruckelnde Bewegung. Zum Vergleich: Die Bildwiederholfrequenzen der Fernsehmonitore liegen bei 50 Hz, die Monitore der für Graphik genutzten Computer zwischen 75 bis 120 Hz bei CRTs (und 50 bis 75 Hz bei TFTs, die allerdings variabel eingestellt werden kann). Erst bei dieser Wiederholrate wird ein Standbild auf einem CRT als ruhig wahrgenommen. Bei bewegten Bildern ist das Auge oder Gehirn toleranter. Trotzdem möchte man die schnelle Taktung der Monitore nicht an die Zeichenalgoritmen der Computergraphik verlieren. Der einfache Trick besteht darin, den *Colorbuffer* zu verdoppeln, wobei immer der *Frontbuffer* dargestellt wird. Gezeichnet wird immer im *Backbuffer*. Durch das Umlegen eines einzelnen Bits wird der *Backbuffer* nach vorne geholt und zum *Frontbuffer* erklärt, während der frei gewordene Speicher des bisherigen *Frontbuffers* zum erneuten Zeichnen genutzt werden kann. Das Tauschen der Buffer geschieht mit einem einzigen Befehl. Da hier die Größe des Graphikfensters eingeht, lautet der Aufruf der X-Routine für X-Windowsysteme

```
glXSwapBuffers(Display *dpy, GLXDrawable drawable);,
```

wobei *dpy* die Verbindung zum X-Server herstellt und *drawable* das Fenster spezifiziert, auf das sich der Buffertausch bezieht. Da innerhalb der `GLUT` dem Nutzer die X-Serveranbindung abgenommen wird und immer in das letzte aktive Fenster gezeichnet wird, genügt hier am Ende der Zeichenroutine ein Aufruf

```
glutSwapBuffers();.
```

Dieser Befehl wird einfach nach jedem vollständigen Zeichnen eines *Frames* aufgerufen und tauscht die *Buffer*. Jedes Neuzeichnen beginnt immer mit einem Löschen des vorhandenen Inhalts des *Buffers*.

Will man wirkliches dreidimensionales Sehen erfahrbar machen, muss für jedes Auge ein Bild aus dem Blickwinkel dieses Auges errechnet werden. Soll im Stereomodus gearbeitet werden, braucht man für eine Animation jeweils die Buffer *Front Left* und *Front Right* sowie *Back Left* und *Back Right*, wobei mit

```
glDrawBuffer(GL_BACK_RIGHT);
```

beispielsweise der *Backbuffer*, der für das rechte Auge bestimmt ist, zum Zeichnen aktiviert wird. Die Begriffe *Passiv* und *Aktiv Stereo* beziehen sich auf die technische Umsetzung in der Hardware.

4.1.1 Passiv Stereo für räumliches Sehen

Passiv Stereo meint die passive Trennung der Information für die verschiedenen Augen über Filter, die sowohl den Projektoren als auch den Brillen vorgeschaltet werden. Bei dem rot-grün Verfahren arbeitet man mit entsprechenden grün-roten Brillen und macht sich den Komplementärkontrast zunutze, der die ausgewogenste Helligkeitswahrnehmung des Menschen für die beiden Farbkanäle hat (die Komplementärkontraste blau-orange oder gar violett-gelb verlagern die Wahrnehmung sehr viel stärker ins orange bzw. gelbe Spektrum). Klarerweise verliert man mit diesem Verfahren die Farbinformation zugunsten der räumlichen Information der dargestellten Szene, da die Farbe zur Kanaltrennung eingesetzt wird.

Will man die volle Farbtiefe beim Passiv Stereo erhalten, setzt man Polarisationsfilter ein. Bei den Projektionsverfahren, die mit horizontal und vertikal polarisiertem Licht (und entsprechenden Brillen) arbeiten und mit denen man bei Großprojektionen in den Kinos oder auf sogenannten Stereoboards verfährt, wird jetzt zwar die Farbinformation erhalten, aber der Betrachter darf seinen Kopf nicht zur Seite neigen. Bei 45° Neigungswinkel sind die Brillen maximal unwirksam und man sieht das projizierte Doppelbild statt des Stereoeffekts. Kippt man den Kopf bis zu 90°, ist das Gehirn erstaunlicherweise in der Lage, die rechts-links Information, die jetzt auf ein oberes und unteres Auge gegeben wird, als räumliches Bild zu interpretieren; erstaunlich deshalb, weil die für das linke Auge berechnete Szene jetzt für das rechte Auge sichtbar ist und umgekehrt.

Abhilfe schaffen hier Projektoren, die zirkular polarisiertes Licht ausstrahlen. Mit entsprechenden Filtern versehene Brillen trennen die Bilder bei jeglicher Drehung des Kopfes, da die Symmetrie dabei erhalten bleibt (konzentrische Kreise für ein Auge, konzentrische Strahlen für das andere Auge). Allerdings ist die Kanaltrennung nicht so sauber, so dass man Doppelbilder sieht.

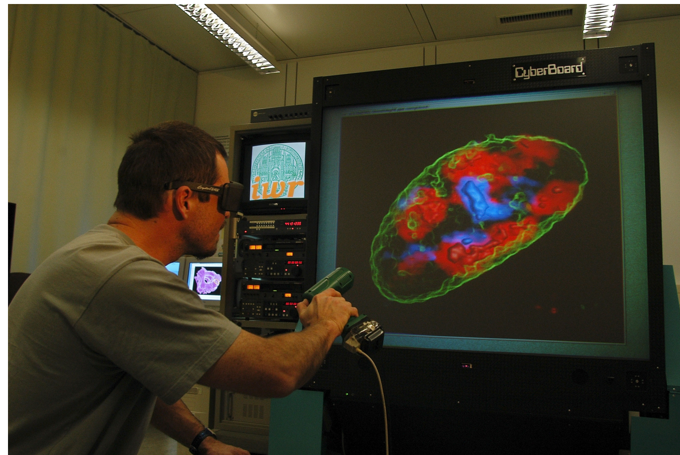


Abbildung 4.2. Chromosomendomänen in einer Eizelle am *Stereoboard* im Interdisziplinären Zentrum für Wisennschaftliches Rechnen (IWR) der Universität Heidelberg.

4.1.2 Aktiv Stereo für räumliche Interaktionen

Aktiv Stereo bietet sich einerseits für Arbeitsplatz-Installationen an, die dazu keine Projektoren mit Projektionsmattscheibe sondern einen stereofähigen (Röhren-)Monitor, einen Infrarotsender und eine spezielle Brille brauchen. Über den Sender wird eine sogenannten *Shutterbrille* mit dem Schirm gekoppelt, so dass abwechselnd ein Bild für das linke, dann das rechte Auge darstellt werden kann, und in gleicher Geschwindigkeit das rechte und dann das linke Glas der Brille verschlossen wird. Bei der Darstellung von hochkomplexen Makromolekülen, deren räumliche Struktur und dynamische Bewegung sehr viel über ihre Funktionsweise aussagen, kommt entsprechende Soft- und Hardware zur Anwendung.

Ein weiteres Einsatzgebiet für diese technisch aufwendige Lösung findet sich in größeren *Virtual Reality Installationen*. In einer sogenannten *Cave* wird mit (Rück-)Projektionen auf die Wände und den Boden einer Raumecke oder gar eines Würfels gearbeitet. Für die Nutzer, die sich in einer solchen Installation befinden, erscheinen die gerenderten Bilder jetzt durch die *Shutterbrille* betrachtet als im Raum stehende Lichtobjekte. Kein Rand eines Displays, kein Fensterrahmen beschneidet den 3D-Eindruck. Dazu werden die aktiv-stereofähigen Projektoren wieder über Infrarotsender mit den *Shutterbrillen* gekoppelt. Zudem arbeitet man in einer *Cave* mit einem *Tracking System*, das die Augenposition des Betrachters und die Position eines 3D-Pointers feststellt. Diese Positionen können über Ultraschall, Infrarot, Trägheitssensoren oder, besonders stabil(!), über optische Methoden gemessen werden. Bei den optischen Methoden verfolgen mehrere Kameras auffällig markierte Raumpunkte an einer dadurch ausgezeichneten Brille (und damit einem ausgezeichneten Betrachter) bzw. am 3D-Pointer. Mit Bilderkennungsverfahren werden die räumlichen Positionen errechnet, um die entsprechenden Szenen dafür passend rendern zu können.

Diese Installationen erlauben die beliebige Bewegung dieses einen Betrachters und Interaktionen mit dem Objekt über den 3D-Pointer. Typische Einsatzgebiete einer Cave finden sich im Bereich der Ent-

Abbildung 4.3. Interaktion bei simulierter Signalübertragung in Nervenzellen in der *Cave* am Höchstleistungsrechenzentrum der Universität Stuttgart (HLRS) mit optischem Trackingsystem.

wicklung von Prototypen in der Produktfertigung (Fahrzeugbau). Hier lohnt sich die Anschaffung, da man vor der Erstellung eines realen Prototypen viele virtuelle (beispielsweise im simulierten Windkanal) testen und verwerfen kann.

4.2 Depth- oder z-Buffer

Bei voreingestellter Kamerasicht auf eine Szene sind x- und y-Koordinaten nur horizontale oder vertikale Versetzungen. Der z-Wert enthält den Abstand vom Auge des Betrachters senkrecht zum Schirm. Die Wahrnehmung eines dreidimensionalen Objekts wird erheblich erleichtert, wenn weiter hinten liegende Linien und Flächen durch die dem Auge näheren verdeckt werden. Eine elegante Methode nutzt den sogenannten Tiefenspeicher oder auch *Depth-* oder *z-Buffer*. Diese Idee wird meist mit dem Namen Ed Catmull assoziiert (siehe Abbildung 4.4). Für jeden Pixel wird ein z-Wert abgelegt. Die grundsätzliche Idee ist nun, einen Test auf den darin enthaltenen Wert zu starten. Für jedes neu zu zeichnende Objekt wird für jeden am Zeichnen beteiligten Pixel ein z-Wert ermittelt. Je größer dieser Wert ist, desto weiter ist das Objekt vom Betrachter entfernt. Nur wenn der z-Wert geringer ist, als der für diesen Pixel bereits gespeicherte Wert, wird der Farbwert dieses Pixel erneut ermittelt und im *Colorbuffer* eingetragen.

Für die Umsetzung fügen Sie dem Aufruf

```
glutInitDisplayMode(...|GLUT_DEPTH|...);
```

Abbildung 4.4. Edwin Catmull (1945*) bei der Verleihung der *NYIT President's Medal for Arts and Technology*, 2001.

hinzu und aktivieren den z-Buffer mit

```
glEnable(GL_DEPTH_TEST);
```

ACHTUNG! Da Animationen ein permanentes Neuzeichnen nötig machen, müssen diese Buffer zu Beginn immer wieder gelöscht werden. Bei hochauflösenden Schirmen und einer maximalen Fenstergröße heißt das, 1 280 × 1 024 = 1 310 720 Pixel müssen einzeln angesteuert und in den verschiedenen Buffern zurückgesetzt werden. Dieses Löschen ist eine der teuersten Operationen innerhalb des Darstellungsprozesses.

Wenn von der Hardware das gleichzeitige Zurücksetzen mehrerer Buffer unterstützt wird, kann der *OpenGL* Befehl

```
glClear(...|GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT|...);
```

das Zurücksetzen entsprechend optimieren. Diese Art des Löschens kann auch auf weitere Buffer ausgedehnt werden. Ansonsten werden die Speicher nacheinander zurückgesetzt.

Mit dem Befehl

```
glClearColor(GL_FLOAT r, GL_FLOAT g ,GL_FLOAT b, GL_FLOAT α);
```

kann eine Hintergrundfarbe (Voreinstellung: Schwarz), und mit

```
glClearDepth(GL_FLOAT depth);
```

eine maximale Tiefe (Voreinstellung: 1, Zwangsbedingung `depth` $\in [0, 1]$) vereinbart werden.

Eine historische Anmerkung zur veralteten *GL*-Bibliothek auf *SGI*-Maschinen: die Befehle `clear(0xff969696);` und `zclear(0xffffff);` werden dort zu `czclear(0xff969696,0xffffff);` zusammengefasst. Das `c` steht dabei für *Color*, das `z` steht für den z-Wert, und die beiden Hexadezimalzahlen geben die Farbe (ein helles Grau) und den maximalen z-Wert explizit an.

Abbildung 4.5. Wolfgang Straßer (1945*), Leiter des Wilhem Schickard Instituts für Informatik, Graphische Interaktive Systeme (GRIS) an der Universität Tübingen.

4.2.1 Der Depth Buffer Algorithmus

Der ungefähr zeitgleich von Edwin Catmull in Utah und Wolfgang Straßer in seiner Dissertation 1974 an der TU Berlin entwickelte z-Buffer Algorithmus wird gemeinhin dafür genutzt, Farbwerte verdeckter Pixel erst gar nicht zu berechnen.

`Pseudocode`:

```
void zBuffer(int pz)
{
  int x, y;
  for (y = 0; y < YMAX; y++){
    for (x = 0; x < XMAX; x++){
      WritePixel(x,y, BACKGROUND_COLOR);  /* Clear color */
      WriteZ(x,y, DEPTH);                 /* Clear depth */
    }
  }
  for(eachPolygon){
    for(eachPixel in polygon's projection){
      pz = polygon's z-value at pixel's coordinates(x,y);
      if (pz <= ReadZ(x,y)){              /* New point is nearer */
        WriteZ(x,y,pz);
        WritePixel(x,y, polygon's color at (x,y));
      }
    }
  }
}
```

Ob ein Punkt im Colorbuffer dargestellt wird, entscheidet der Wert, der im z-Buffer eingetragen ist. Pixel mit großem z-Wert werden von Pixeln mit kleinerem z-Wert <u>überschrieben</u>.

Die Vergleichsoperationen können allerdings verändert werden, indem man

```
glDepthFunc(GLenum func);
```

mit einer der folgenden Indikatoren für die Operatoren aufruft: `GL_LESS, GL_LEQUAL, GL_GREATER, GL_NEVER, GL_ALWAYS, GL_EQUAL.`

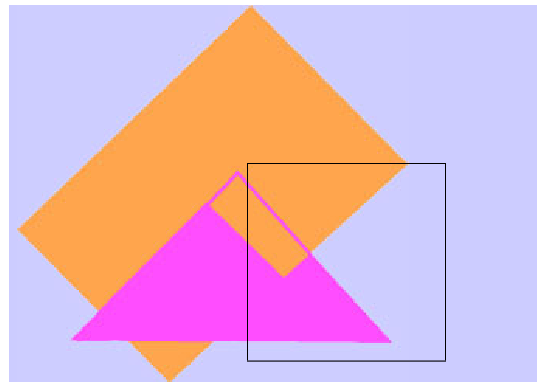


Abbildung 4.6. Die vollständige Szene und der Fensterausschnitt.

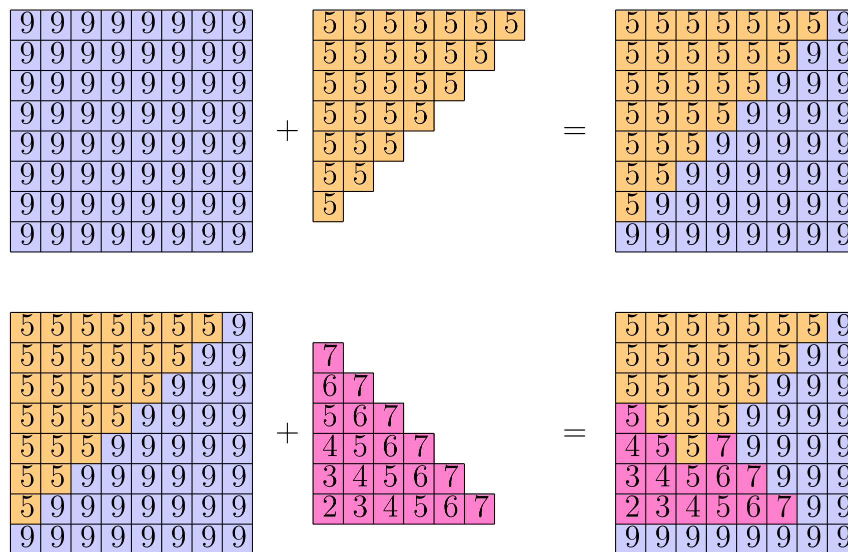


Abbildung 4.7. Der Wert im z-Buffer entscheidet über die Darstellung des Pixels.

Die Vorteile des z-Buffer Algorithmus bestehen darin, dass kein Vorsortieren der Objekte nötig ist und auch während der Transformationen wie Drehungen kein Vergleichen von Objekten gemacht werden

muss. Die Polygone erscheinen in der Reihenfolge, in der sie gezeichnet werden, aber aufwendiges Rendern von Farbwerten verdeckter Pixel wird eingespart. Zur Illustration zeigt Abbildung 4.6 ein pinkfarbenes Dreieck, das von schräg vorne ein oranges Rechteck vor einem hellblauen Hintergrund durchsticht. Abbildung 4.7 stellt nacheinander die im Pseudocode vorgestellten Operationen dar. Zunächst wird der Colorbuffer, der in der Größe des Fensters angelegt wurde, gelöscht. Es werden Farb- und z-Werte zurückgesetzt. Dann wird der z-Wert des orangen Ausschnitts des Rechtecks in den z-Buffer eingetragen und der Farbwert in den Colorbuffer übernommen. In der unteren Bildhälfte werden diejenigen Pixel der Projektion des zweiten Polygons in das Fenster eingetragen, deren Wert im z-Buffer geringer oder gleich den vorhandenen Werten ist. Die Vergleichsoperation ist also `GL_LEQUAL`.

4.2.2 Das Zeichnen von Polygonen aus Dreiecken

Die auf den Schirm projizierten Polygone werden entlang der sogenannten *Scanline* gezeichnet. Alle Polygone sind zunächst planar und aus Dreiecken zusammengesetzt. Es treten je nach Implementierung der Zeichenroutinen Fehler auf, wenn nicht konvexe, nicht ebene Polygone unspezifisch angegeben werden, da die Graphiksysteme intern verschiedene Algorithmen zur Triangulierung benutzen.

Zur Bestimmung des z-Wertes muss eine Ebenengleichung gelöst werden.

$$Ax + By + Cz + D = 0 \qquad \Rightarrow \qquad z = \frac{-Ax - By - D}{C}$$

Die entlang der *Scanline* gefundenen z-Werte werden jetzt linear interpoliert (siehe Abbildung 4.8).

$$\left.\begin{aligned} z_a &= z_1 - (z_1 - z_2)\frac{y_1 - y_s}{y_1 - y_2} \\ z_b &= z_1 - (z_1 - z_3)\frac{y_1 - y_s}{y_1 - y_3} \end{aligned}\right\} \qquad \Rightarrow \qquad z_p = z_b - (z_b - z_a)\frac{x_b - x_p}{x_b - x_a}$$

Zur Beschleunigung des Algorithmus kann der vorherig berechnete z-Wert $z(x, y)$ benutzt werden.

$$z(x + \Delta x, y) = z(x, y) - \frac{A}{C}(\Delta x)$$

Typischerweise ist das Inkrement $\Delta x = 1$ und $A/C \equiv$ const. So ergibt sich eine Subtraktion für jedes neue Pixel entlang der Scanline.

$$z(x + 1, y) = z(x, y) - \text{const.}$$

Gleiches gilt für eine inkrementelle Berechnung des ersten z-Werts in der nächsten Scanline, der um B/C vermindert werden muss.

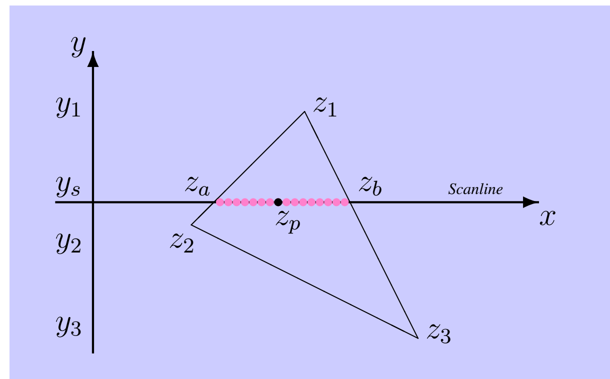


Abbildung 4.8. Der z-Wert wird linear interpoliert.

Ein gewisser Gewinn an *Performance* kann durch grobes Vorsortieren erzielt werden. Wenn das Rendern eines Objekts, d. h. die Bestimmung der Farbwerte der beteiligten Pixel, aufwendig ist, lohnt sich ein Sortieralgorithmus nach z-Werten, bevor mit dem z-Buffer gearbeitet wird. So wird das vorderste Objekt vollständig gerendert und von den weiter hinten liegenden nur die Teile, die den z-Test bestehen, also seitlich hinter dem ersten Objekt hervorschauen. Dieses Vorsortieren muss besonders bei Drehungen in regelmäßigen Abständen wiederholt werden.

Merksatz: Der z-Buffer ist pixelorientiert und nicht an Objekte gebunden.

Die Anzahl der Objekte beeinflusst die Geschwindigkeit des Algorithmus kaum, da mit steigender Anzahl sich immer mehr Objekte gegenseitig überdecken und außerdem die Größe der Objekte abnimmt. (Einzig über einen gewissen *Scan-conversion overhead* verlangsamt sich die Auswertung.)

Probleme macht allerdings das korrekte Darstellen von Transparenz. Hier ist das zusätzliche Speichern objektorientierter z-Werte nötig, da die Reihenfolge des Zeichnens über die korrekte Darstellung entscheidet.

4.2.3 Hidden Surface

In *OpenGL* werden undurchsichtige Objekte bei ***aktiviertem*** und vor Zeichenbeginn ***geleerten*** z-Buffer automatisch korrekt gezeichnet. Das einzige Problem taucht bei annähernd gleich großen z-Werten auf. Wenn sich Dreiecke unter sehr geringem Neigungswinkel durchdringen oder gar zwei Dreiecke unterschiedlicher Farbe direkt aufeinander liegen, kann man die *Scanlines* erkennen. Beinahe zufällig gewinnt die eine oder die andere Objektfarbe bei der *Scan Conversion*, also dem Projizieren der Dreiecke. Diesen Effekt bezeichnet man als ***z-Buffer Conflict***. Insbesondere beim *Zoomen* einer Szene entsteht ein störendes Flackern dieser Dreiecke. Letztlich sind dafür Rundungsfehler im z-Buffer verantwortlich, die einen der beiden übereinanderliegenden Farbwerte zufällig begünstigen.

4.2.4 Hidden Line

Wenn Sie ein Drahtgittermodell mit aktiviertem z-Buffer zeichnen, so werden nur die ***Schnittpunkte*** der Linien als Objekte aufgefasst. Bei einer üblichen Linienstärke von 1 handelt es sich also um einzelne Pixel, die kaum wahrnehmbar sind. Deshalb lässt sich bei einem derartigen Drahtgittermodell häufig schlecht erkennen, was vorne, was hinten ist. Bei Handzeichnungen behilft man sich, indem man hinten liegende Linien gestrichelt darstellt. Das geht in der Computergraphik nicht so einfach, da eine verdeckte Linie durch Drehen durchaus wieder in den Vordergrund geraten kann. Dabei ergeben sich entlang derselben Linie immer wieder neue Schnittpunkte mit verdeckenden Flächen. Jedesmal müsste der Linienstil an der entsprechenden Stelle geändert werden, as nicht praktikabel ist. Stattdessen setzt man sogenannte atmosphärische Effekte ein. Man stellt die Linien in Abhängigkeit ihres z-Werts automatisch mit geringerer Intensität dar (*FOG* (Nebel, Dunst) oder *Depth Cueing*, siehe Abschnitt 5.4.5).

Die *Hidden Line* Darstellung meint, dass man diese Linien gar nicht darstellt. Dazu zeichnet man das Objekt im Polygonmodus `glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);` z. B. in der Hintergrundfarbe und im Polygonmodus `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);` in der gewünschten Objektfarbe.

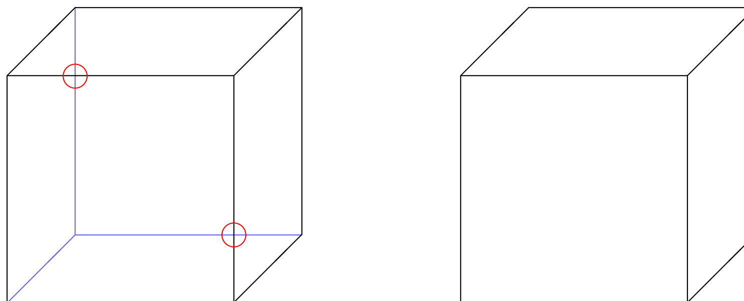


Abbildung 4.9. Das Drahtgitter eines Würfels und seine *Hidden Line* Darstellung.

Da Polygone und Linien sehr verschiedenen Konversionsalgorithmen unterliegen, tritt hierbei typischerweise ein dem z-Buffer Conflict ähnliches Problem auf (siehe Abschnitt 4.2.3). Entlang der Polygonkanten haben die berandenden Linien und die ausfüllenden Flächen nahezu gleiche z-Werte. Sie sorgen dafür, dass einmal die Linie, dann wieder das Polygon bei leichter Drehung den z-Test für denselben Pixel gewinnt. Dieser Effekt erzeugt immer wieder in der Linie fehlende Pixel, so dass die Linie unterbrochen wie gestrichelt erscheint. Daher kommt auch der Name *Stitching* für dieses Phänomen.

Als Abhilfe addiert man einen sogenannten *Polygonoffset*, um die z-Werte sauber voneinander zu trennen.

```
glEnable(GL_POLYGON_OFFSET_FILL);
glEnable(GL_POLYGON_OFFSET_LINE);
```

```
glEnable(GL_POLYGON_OFFSET_POINT);
```

Der Wert des *Polygonoffset* wird nach der Formel

$$o = m \cdot factor + r \cdot units$$

jeweils neu berechnet. Dabei ist m der maximale Tiefengradient des Objekts und r der kleinste Wert, der für die Darstellung auflösbare z-Werte erzeugt. Der letztere Wert ist implementierungsabhängig respektive durch die Hardware bedingt und beschreibt, wie das Interval $[0.0, 1.0]$ auf den *z-Range* $[0, \mathtt{ZMAX}]$ abgebildet wird. Die Größen $factor$ und $units$ werden vom Anwendungsprogramm vorgegeben und sind darstellungsabhängig. Der *OpenGL*-Aufruf lautet jetzt:

```
glPolygonOffset(GLfloat factor, GLfloat units);
```

Einfachste Werte von $factor$ und $units$ sind 1.0. Das ist unter Umständen aber nicht hinreichend. Wenn die Linienstärke beispielsweise größer als 1.0 ist, sollte man den $factor$ erhöhen. Auch bei perspektivischen Projektionen kann man sich überlegen, den $factor$ an den z-Wert zu koppeln (der Offset muss erhöht werden, wenn das Objekt weiter entfernt wird).

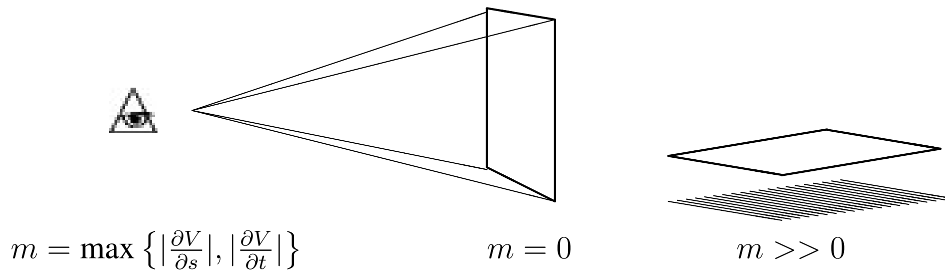


Abbildung 4.10. Die Neigung m ist objekt- und projektionsabhängig.

Alternativ kann dabei entweder $o > 0$ sein, dann addiert man einen positiven Offset zum gefüllten Objekt, es erscheint weiter hinten. Oder $o < 0$, also addiert man einen negativen Offset zum Linienobjekt, wobei es weiter vorne erscheint.

Allgemein gilt, besser zuviel Offset als zuwenig zu addieren! Das *Stitching* ist bei weitem auffälliger als die sogenannten *Sternchen* an den Eckpunkten, wo die Linien zusammenkommen. Bei ungünstigem Blickwinkel kann das füllende Objekt durchaus kleiner und neben dem Linienobjekt erscheinen und daher die abgewandten Linien nicht vollständig überdecken.

Implementierung des *Hidden Line*-Algorithmus mit Poygonoffset:

```
glEnable(GL_DEPTH_TEST);
glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);  /* outline */
    set_color(foreground);
    draw_object_with_filled_polygons();
glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);  /* opaque object */
glEnable(GL_POLYGON_OFFSET_FILL);
  glPolygonOffset(1.0,1.0);                /* push it away */
    set_color(background);
    draw_object_with_filled_polygons();
glDisable(GL_POLYGON_OFFSET_FILL);
```

Die gesonderte Darstellung eines Objekts und seiner Kanten wird auch zum Glätten *(Antialiasing)* der Kanten oder zum besonderen Hervorheben mit geänderten Materialeigenschaften verwendet (spiegelnde, aufblitzende Kanten im Gegensatz zu matten Oberflächen). Eine weitere Möglichkeit einer *Hidden Line* Realisierung, die sehr viel korrekter arbeitet, aber dafür auch extrem langsam ist, wird mit dem im nächsten Abschnitt behandelten *Stencil Buffer* betrieben.

4.3 Stencil Buffer

Der *Stencil Buffer* oder Schablonenspeicher dient zum Markieren der Bereiche, die einen weiteren Bearbeitungsschritt erfahren sollen. Wenn beispielsweise bei einem Flugsimulator nur das Cockpitfenster und wenige Instrumente neu gezeichnet werden müssen, so maskiert man diese Bereiche. In gleicher Weise kann man auch die Bereiche markieren, die ein Linienobjekt annimmt, und so einen *Hidden Line*-Algorithmus implementieren.

Implementierung des *Hidden Line*-Algorithmus mit dem *Stencil Buffer*:

```
glEnable(GL_STENCIL_TEST);
glEnable(GL_DEPTH_TEST);
glClear(GL_STENCIL_BUFFER_BIT);
for (i=0; i< max; i++){
  glStencilFunc(GL_ALWAYS,0,1);
  glStencilOp(GL_INVERT,GL_INVERT,GL_INVERT);
  set_color(foreground);
  outline_polyon(i);       /* outline */
  glStencilFunc(GL_EQUAL,0,1);
  glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
  set_color(background);
  filled_polygon(i);       /* inner object */
}
```

Grundsätzlich gilt, dass ein *Hidden Line*-Algorithmus mit Polygonoffset immer schneller ist, während der *Stencil Buffer* dafür korrekter arbeitet.

4.4 Accumulation Buffer

Der *Accumulation Buffer* arbeitet wie ein vervielfältigter Colorbuffer. Man sammelt hier viele einzelne Frames, bevor sie in gewünschter Weise zusammengefasst und an den Colorbuffer übergeben werden. Der Begriff des *Jittering* = Zittern wird für diese Art des Zeichnens auch oft verwendet. Die RGBA-Werte des Farbspeichers werden leicht gegeneinander versetzt, um aus dieser Folge von Bildern ein Bild zusammen zu setzen. In der Regel wird dadurch ein verschwommenes Bild erzeugt (*Blur Effect*), das auch in einer ausgezeichneten Richtung verschmiert erscheinen kann (*Motion Blur*). Es können aber auch Bilder aus leicht unterschiedlicher Perspektive gerendert werden, was beim Interpolieren sogenannte Linseneffekte bewirkt.

Bilder aus dem Accumulation Buffer können einfach linear interpoliert werden, in dem man die Farbwerte von allen Bildern an einem Speicherplatz (Pixel) addiert und durch die Anzahl der Bilder teilt. Aber auch gewichtetes, nichtlineares Interpolieren kann sinnvoll sein, wenn man dynamische Effekte erzeugen will, die in einem Standbild festgehalten werden sollen. Hier wird man zeitlich frühere Bilder geringer wichten wollen, als weiter an der Endeinstellung gelegene Projektionen.

Es gibt also folgende Verwendungsmöglichkeiten des Accumulation Buffer:

1. *Antialiasing* komplexer Szenen
2. *Motion blurred* (bewegungsverzogene) Objektdarstellungen
   (a) Dynamische Vorgänge in Standbildern (e. g. Sportfotografie)
   (b) Dynamische Vorgänge in animierten Sequenzen, die den Bewegungsablauf in Form eines schwindenden Bildgedächnisses speichern (bewegungsverzogene Animationen)
3. Stereoeffekte in rot/grün Manier

Das Implementieren ist etwas aufwendiger und die Rechenzeiten sind so groß, dass der Accumulation Buffer für alle in Echtzeit zu rendernden Animationen leider kaum sinnvoll einsetzbar ist. Dennoch sind die Ergebnisse oft so reizvoll, dass sich der Aufwand lohnt, Filme aus derart erzeugten einzelnen Frames zusammen zu setzen.

Implementierung des *Scene Antialiasing* mit dem Accumulation Buffer:

```
glClearAccum(background);
glClear(...|GL_ACCUM_BUFFER_BIT|...);
for (jitter=0; jitter < ACSIZE; jitter++){
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT|...);
  accPerspective(...);    /* user supplied accumulation */
                          /* of slightly translated pictures */
  displayObjects();
  glAccum(GL_ACCUM,1.0/ACSIZE);
} glAccum(GL_RETURN,1.0);
glFlush();
```

4.5 Abfolge der Testoperationen

Ein in *OpenGL* generiertes und mit Farbwerten versehenes Objekt durchläuft in der folgenden Reihenfolge weitere Tests, bevor die Pixel des *Colorbuffer* erreicht werden:

1. *Scissor Test* entscheidet über neu zu zeichnende rechteckige Bildschirmausschnitte.
2. *Alpha Test* sagt, ab welchem Schwellwert Durchsichtigkeit nicht mehr Zeichnen bedeutet.
3. *Stencil Test* entscheidet über irregulär geformte Bereiche und die Art, wie darin zu zeichnen ist.
4. *Depth Test* entscheidet nach dem z-Wert, ob dieses Pixel gezeichnet wird.
5. *Blending* überschreibt oder verrechnet mit dem Hintergrund- oder Nebelfarbton.
6. *Dithering* erzeugt Mischfarben.
7. *Logical Operations* sind weitere Boolsche Operationen auf die gespeicherten Objekte *(e.g. Clipping)*

4.6 Übungsaufgaben

Aufgabe 4.1 Animation

Schreiben Sie eine kleine Animation (aktivieren Sie dazu den Darstellungsmodus `GLUT_DOUBLE` *und arbeiten Sie mit* `glutSwapBuffers();`*). Benutzen Sie die 3D-Objekte aus den Bibliotheken von* `gluSphere(GLUquadric* quad, GLdouble radius, GLint slices, GLint stacks);` *bis* `glutWireTeapot(GLdouble size);` *nach Belieben. Lassen Sie beispielsweise ein 3D-Objekt mit* `glRotated(GLdouble angle, GLdouble x, GLdouble y, GLdouble z);` *um eine Achse drehen. Binden Sie eine* `glutMouseFunc(void (*func)(int button, int state, int x, int y));` *so ein, dass die Animation erst startet, wenn Sie eine Mousetaste drücken, und stoppt, wenn Sie sie loslassen. Schließen Sie jetzt Drehungen mit den verbleibenden Mousetasten um die weiteren beiden Raumachsen an.*

Aufgabe 4.2 Verzerrungen des Raumes

Skalieren Sie Ihr 3D-Object in einer Raumrichtung. Schließen Sie jetzt eine Drehung um eine andere Raumrichtung an (als gesteuerte Animation, siehe Aufgabe 4.1). In welchem Matrixmodus befand sich Ihr Programm während der Transformationen? Was beobachten Sie? Schreiben Sie Ihre Beobachtung als Kommentar in Ihr Programm.

Aufgabe 4.3 Z-Conflict

Erweitern Sie die Szene aus Aufgabe 3.3, indem Sie an eine der Wände ein Polygon mit einer von der Wandfarbe verschiedenen Farbe zeichnen. Achten Sie darauf, dass das Polygon sich in der gleichen Ebene wie die Wand befindet. Ermöglichen Sie auch hier eine Rotation der Szene, sodass sich beide Polygone gemeinsam drehen.

Der Z-Buffer Conflict *erzeugt nun ein auffälliges Muster, das durch konkurrierende z-Werte der verschiedenen übereinandergezeichneten Polygon-Farbwerte erzeugt wird, die entlang einer Scanline abrupt umschalten. Aufgrund von Rundungsfehlern kann das Umschalten in der nächsten Scanline diskontinuierliche Sprünge aufweisen, da es viel früher (oder später) geschieht, als in der vorangegangenen Scanline. Wodurch kann man diesen Effekt möglichst unterdrücken? Ermöglichen Sie ein Umschalten zwischen beiden Versionen über Menü und kommentieren Sie Ihr Vorgehen an den entsprechenden Stellen in Ihrem Programmcode hinter dem Begriff* Z-Conflict.

Aufgabe 4.4 Hidden Lines

Mit einer Hidden Lines *Darstellung meint man, dass bei einem 3D-Drahtgittermodell nur die Linien(teile) zu zeichnen sind, die nicht von den das Gitter ausfüllenden Flächen verdeckt werden. Zeichnen Sie ein beliebiges 3D-Drahtgitter, das sich um eine Achse dreht. Nutzen Sie den z-Buffer für einen Hidden Lines Algorithmus. Ermöglichen Sie, mit dem* `zKey` *nahe an Ihr Objekt heranzufahren, um das Ergebnis überprüfen zu können.*

Aufgabe 4.5 Animieren des Roboterarms

Animieren Sie Ihren Roboterarm aus Aufgabe 2.7, indem Sie die drei Gelenke unabhängig voneinander über die Tasten 1, 2 und 3 bewegen lassen. Bauen Sie für das erste Gelenk einen begrenzten Winkelbereich der Beweglichkeit ein (z. B. Vermeiden des Überstreckens, Vermeiden von Selbstdurchdringung). Arrangieren Sie die Finger so, dass ein Zugreifen mit einer weiteren Taste 4 geschieht.

Kapitel 5

Farbe und Licht

Farbe und Licht gehören auf dem Bildschirm untrennbar zusammen. Wählt man als Ausgabemedium für graphische Inhalte den Drucker, werden die Farben je nach Technik untereinander gemischt und auf ein Trägerpapier aufgebracht. Häufig ist das Ergebnis unerwartet dunkel, da das Papier anders als der Bildschirm nicht von sich aus leuchtet. Die schwierigste Farbe beim Drucken ist blau. Vermeiden Sie außerdem schwarze oder überhaupt dunkle Hintergründe. Auf einem Dia oder mit einem Beamer (in der Projektion mit Licht) mag ein dunkler oder schwarzer Hintergrund noch angehen, da das Bild durch Projektion erzeugt wird und ein ähnlicher Leuchteffekt der Projektionsfläche wie beim Bildschirm das dargestellte Objekt in den Vordergrund rückt. Beim Drucken schluckt das Schwarz enorm viel Farbe und ist daher teuer. Außerdem ist das dunkle Druckergebnis unbefriedigend, denn das Objekt wird dadurch nicht etwa herausgehoben sondern eher nach hinten gedrückt.

Ist das Ausgabemedium ein Video, so hat man bei der Wandlung des Bildschirmsignals in ein Videosignal eine sehr viel geringere Bandbreite der Farbdarstellung zu beachten. Hier sollten Sie satte Farben vermeiden, da sie zum Überstrahlen neigen. Außerdem ist hier das Rot die problematischste Farbe.

5.1 Achromatisches Licht

Achromatisch bedeutet die Abwesenheit von Farbe. Hier geht es nur um die Intensität (Luminanz). Sie ist nicht gleichbedeutend mit Helligkeit. Intensitäten und auch Farben werden heute meist als Gleitkommawerte zwischen 0.0 und 1.0 normiert. Gebräuchlich sind aber auch weiterhin Hexadezimaldarstellungen von Integerwerten zwischen 0 und 255, die bei drei Farbkanälen beispielsweise eine Farbtiefe von 24 bit (3 byte) erzeugen.

0.0 = schwarz (oder in 8 bit (= 1byte) Darstellung 0)
1.0 = weiß (oder in 8 bit Darstellung 255)

Die Intensitätslevel sind logarithmisch, nicht etwa linear gestuft, um gleichmäßige Abstufungen in der Helligkeit zu bewirken. Das trägt der Tatsache Rechnung, dass das menschliche Auge die Intensitätsänderung e.g. von 0.1 zu 0.11 genauso stark wahrnimmt, wie etwa von 0.5 zu 0.55. Eine Glühbirne von 50 Watt gegen eine mit 100 Watt auszutauschen, macht einen erheblich größeren Unterschied, als von 100 auf 150 Watt zu steigern.

5.1.1 Intensitäten

Die Verteilungsfunktion ergibt sich aus folgender einfacher Rechnung:

$$\begin{aligned} I_0 &= r^0 I_0 \\ I_1 &= r^1 I_0 \\ &\vdots \\ I_{255} &= r^{255} I_0 = 1 \qquad \Rightarrow \qquad r = (1/I_0)^{(1/255)} \end{aligned}$$

Damit ergibt sich

$$I_j = r^j I_0 = (1/I_0)^{(j/255)} I_0 = I_0^{(255-j)/255}, \qquad 0 \leq j \leq 255$$

und allgemein für $n+1$ Intensitätsstufen

$$r = (1/I_0)^{(1/n)}, \qquad I_j = I_0^{(n-j)/n}, \qquad 0 \leq j \leq n$$

Beispiel 5.1 *Vier Intensitäten, d. h.* $n = 3$ *bei einer Anfangsintensität von* $I_0 = 1/8 \neq 0$ *ergibt* $r = (1/\frac{1}{8})^{(1/3)} = \sqrt[3]{8} = 2$

$$\begin{aligned} I_0 &= 1/8 \\ I_1 &= 1/4 \\ I_2 &= 1/2 \\ I_3 &= = 1 \end{aligned}$$

Typische Werte für die Anfangsintensität I_0 sind 0.005 oder 0.025. Das ist nicht nur mathematisch bedingt ein von Null verschiedener Wert, sondern beruht auf technischen Gründen.

Bemerkung 5.1 *a) Die minimale Intensität eines Bildschirms kann nicht Null sein, da der Kathodenstrahl* (CRT Cathode Ray Tube)*, durch die Reflexion am Phosphor bedingt, immer eine Resthelligkeit aufweist. Dies ist eine technische Vorgabe, die für jedes Bildschirmmodell gemessen werden muss.*
b) Bei TFTs wird die Hintergrundbeleuchtung meist durch Leuchtstoffröhren oder neuerdings auch superhelle weiße LEDs und Polarisationsfilter realisiert. Ein Bild ergibt sich aus unterschiedlich starker Verdeckung durch Flüssigkristalle. Diese Verdeckung ist nicht vollständig und vor allem an den Rändern schwächer, weshalb ein schwarzes Bild immer eine Resthelligkeit aufweist.

Als *Dynamic Range* bezeichnet man dabei den reziproken Wert der minimalen Intensität:

$$\frac{\text{maximale Intensität}}{\text{minimale Intensität}} = \frac{I_n}{I_0} = \frac{1.0}{I_0} = \textit{Dynamic Range}$$

Messtechisch werden die Intensitäten mit dem Photometer ermittelt, wobei in einem absolut abgedunkelten Raum (zur Vermeidung von Streulicht) auf dem Bildschirm ein weißes Quadrat auf schwarzem Grund erzeugt wird, was typischerweise zu einem *Dynamic Range* von 50 und bei 256 (= 8 bit) Intensitätsstufen zu einem Koeffizienten $r = 1.0154595$ führt.

$$\begin{aligned} I_0 &= 0.02 \\ I_1 &= 0.0203 \\ I_2 &= 0.0209 \\ &\vdots \\ I_{254} &= 0.9848 \\ I_{255} &= 1 \end{aligned}$$

Die γ-(Gamma)Korrektur ist eine Möglichkeit, über *Lookup Tables (LUT)* den tatsächlichen, nichtlinearen Prozess der Verteilung der Intensitäten auf n Stufen CRT-abhängig zu korrigieren.

Ausgabemedium	*Dynamic Range*	Intensitätsstufen bei $r = 1.01$
Diapositiv	1000	700
CRT	50 - 200	400 - 530
SW-Drucker, Fotos	100	465
Farbdrucker	50	400
Zeitungsdruck	10	30

Tabelle 5.1. Intensitäten der verschiedenen Ausgabemedien, aus Foley et al. [FvDF$^+$96a]

5.1.2 Halbtonnäherung

Für *Bilevel-Displays* (Schwarz-Weiß-Schirme) oder 2-3 bit/Pixel Rasterschirme erzeugt man über *räumliche Integration* eine größere Anzahl von Intensitätsstufen, als das Ausgabemedium hergibt. Das menschliche Auge respektive die Bildverarbeitung im Gehirn löst dabei nicht mehr das einzelne Raster auf, sondern fasst größere Bereiche gleichen Musters zu einer Fläche mit entsprechend mittlerer Intensität auf. Diese Technik ist auch unter dem Namen *Halftoning* oder *cluster-dot ordered dithering* bekannt. Der herkömmliche Zeitungsdruck kennt dabei 60-80 verschieden große, unterschiedlich geformte Bereiche pro Zoll. Der Broschürendruck kommt auf 110-200 Bereiche. Als ein einfaches Beispiel für das *Halftoning* denke man an eine Fläche von 2×2 Pixeln, auf der man durch Ein- oder Ausschalten eines Pixels maximal fünf unterschiedliche Intensitätsstufen erzielen kann.

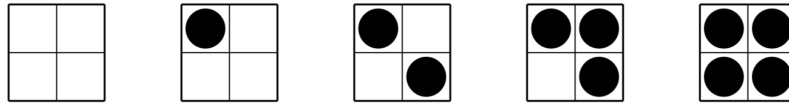


Abbildung 5.1. Fünf Intensitätsstufen auf einer Fläche von 2×2 Pixeln

5.2 Chromatisches Licht und Farbmodelle

Beim Drucken muss man auf die Papierstruktur und die Art des Farbauftrags (matt / hochglänzend) achten, beim Malen kommt es auf Leinwand, Konsistenz der Farbe, Art des Pinsels an. Hier müssen wir uns mit dem Trägermaterial beschäftigen, dessen optische Eigenschaften erst durch das Beleuchten (Ausleuchten) mit einer irgendwie gearteten Lichtquelle einen Eindruck beim Betrachter erzeugt. Bei einem selbst leuchtenden Schirm kann man sich auf die *rein quantitativen physikalischen Eigenschaften* des emittierten Lichts beschränken, also der Farbe einer Lichtquelle, die in diesem Fall ein einzelnes Pixel ist.

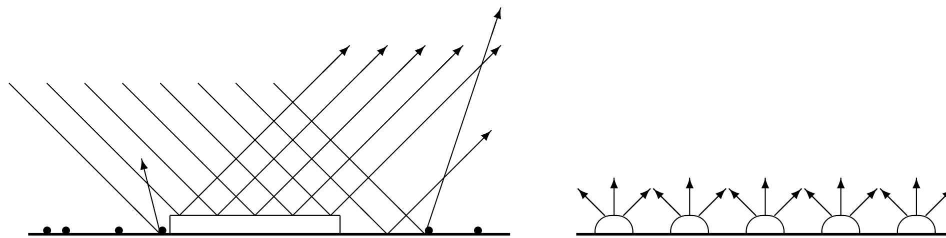


Abbildung 5.2. Beleuchtete Fläche und selbst leuchtender Schirm

5.2.1 Wahrnehmungsmodelle

Definition 5.1 *Ein* Farbmodell *ist ein beschränktes Gebiet in einem 3D Farbkoordinatensystem.*

Definition 5.2 *Dem menschlichen Auge wahrnehmbares* Licht *ist elektromagnetische Energie im Wellenlängenbereich zwischen 400 und 700 nm.*

Die Helligkeitswahrnehmung des menschlichen Auges erreicht ihr Maximum bei 550 nm, was einem Farbwert im gelb-grünen Bereich entspricht. Nach der Tristimulustheorie arbeitet das durchschnittliche menschliche Auge (und damit verbunden die Farberkennung im Gehirn) mit drei verschiedenen Arten von Farbzäpfchen der Retina, die für die Primärfarben rot, grün und blau empfindlich sind.

Das CIE-Chromatische Diagramm von 1931 geht auf die ***C**omission **I**nternationale de l'**É**clairage (CIE)* zurück, die drei Standardprimitive als Integrale über die Energieverteilung definiert. Damit entsteht in einem dreidimensionalen Simplex ein Kegel der sichtbaren Farben.

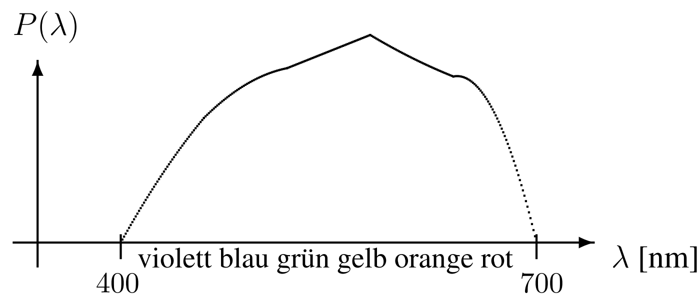


Abbildung 5.3. Energiedichte $P(\lambda)$

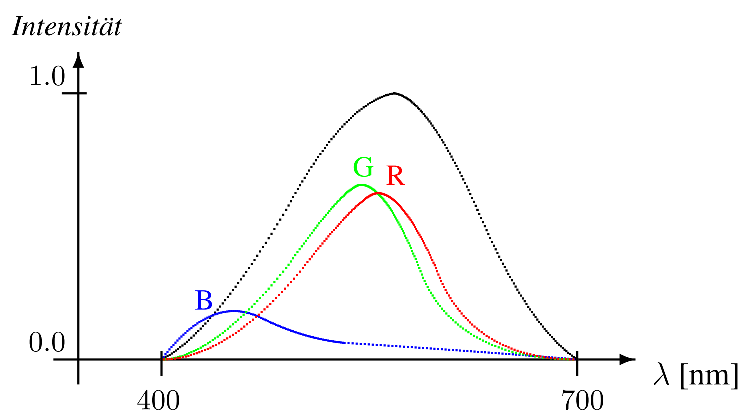


Abbildung 5.4. Intensität und Spektralantwortfunktionen der menschlichen Retina

$$
\begin{aligned}
X &= k \int P(\lambda)\bar{x}_\lambda d\lambda \\
Y &= k \int P(\lambda)\bar{y}_\lambda d\lambda \\
Z &= k \int P(\lambda)\bar{z}_\lambda d\lambda
\end{aligned}
$$

Hierbei stellt $P(\lambda)$ die Energieverteilung, $\bar{x}_\lambda$, $\bar{y}_\lambda$ und $\bar{z}_\lambda$ die jeweiligen Farbfunktionen dar. Der Koeffizient k ist bei einer Normierung auf $Y = 100$ (Weißlicht) gerade 680 Lumens/Watt.

Diesen 3D-Simplex (siehe Abbildung 5.7) stellt man meist entlang der z-Achse projiziert dar, wobei im Schwerpunkt des Dreiecks der sogenannte Punkt *Illuminant C* liegt, der das Weißlicht darstellt (siehe Abbildungen 5.8, 5.9). Am Außenrand des sichtbaren Bereichs kann nun die jeweilige Wellenlänge aufgetragen werden. Für eine beliebige Farbe im Innern gilt, dass die dominante Wellenlänge dieser Farbe durch den Schnittpunkt eines Strahls von *Illuminant C* durch diesen Farbpunkt auf dem Rand gegeben ist. Auch komplementäre Farben lassen sich auf einem in die entgegengesetzte Richtung weisenden Strahl ablesen.

Eine Farbe definiert sich über die Summendarstellung

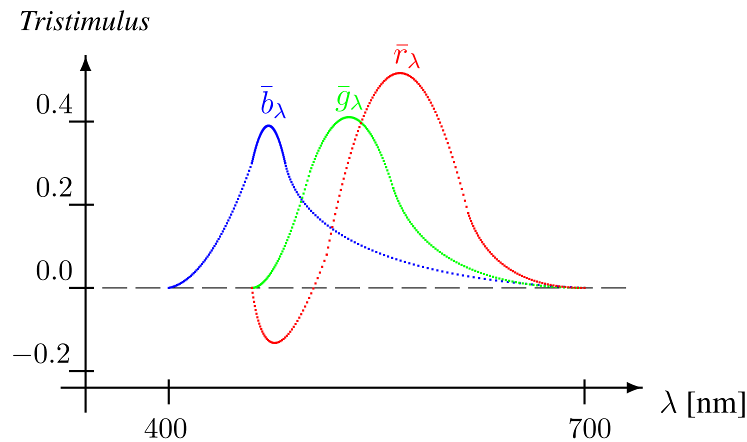


Abbildung 5.5. Tristimuluswerte

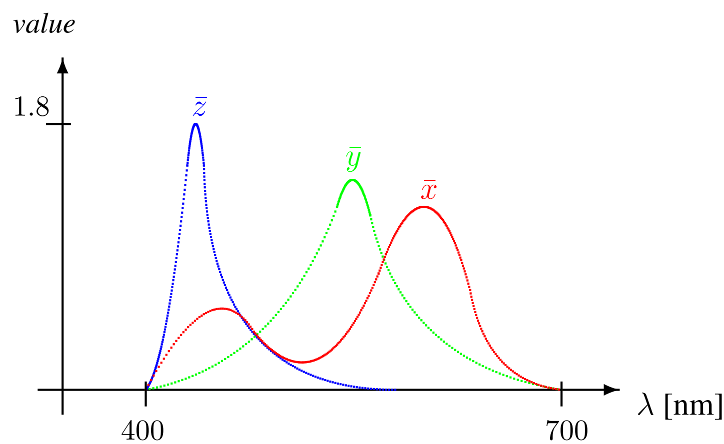


Abbildung 5.6. Primitive

$$C = xX + yY + zZ$$

wobei $x,\ y$ und z integrale Koeffizienten sind und sich in der Summe $x+y+z=1$ zu Eins addieren. Y nennt man auch die Luminanz, Brillanz oder Leuchtkraft. Zero Luminanz lässt sich beispielsweise mit RGB als $0 = 0.177R + 0.812G + 0.011B$ wiedergeben.

Etwas anschaulicher ist das HSV- resp. HSB-Modell[1], dass häufig in Form eines Hexagons gezeigt wird, wobei in der dritten Dimension darunter ein Kegel geformt wird, der den Farbwert (***V**alue*), genauer seine Intensität als Funktion seiner Helligkeit (***B**rightness*) anzeigt.

Das HSV-Modell geht auf Alvy Ray Smith (1978) zurück, wird häufig über Polarkoordinaten angegeben und statt im Hexagon im Kreis angeordnet. Der Farbton (**H**ue) wird deshalb häufig auch mit Farbwinkel übersetzt. Gibt man die Punkte Schwarz und Weiß auf der Werteachse an und blickt von einer geschickten Perspektive auf dieses 3D-Modell, so wird die RGB-Analogie offensichtlich.

[1] Hiermit sind weder der Hamburger Sportverein noch die Heidelberger Straßen- und Bergbahn AG gemeint.

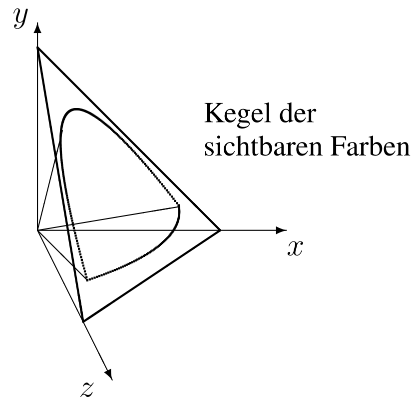


Abbildung 5.7. Simplex

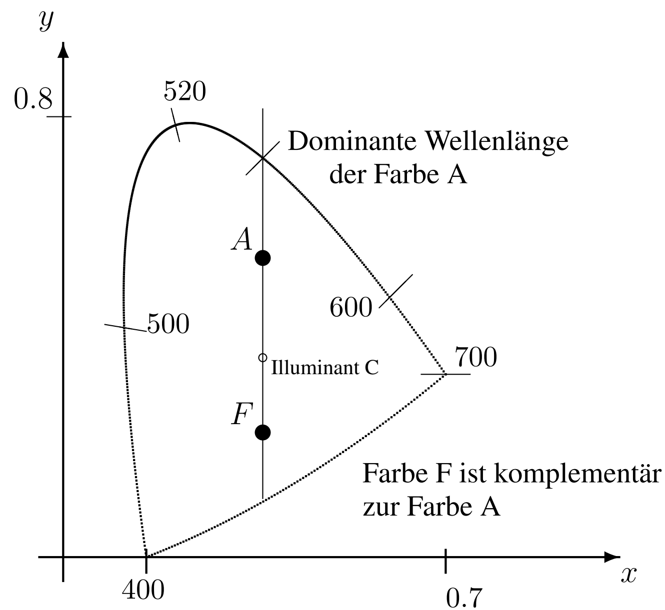


Abbildung 5.8. Projektion auf die $x-y-$Ebene

Bemerkung 5.2 *Die Darstellung als Würfel ähnelt dem Hexagon aus einer bestimmten Perspektive,* aber *lineare Farbinterpolationen innerhalb des RGB-Würfels ergeben nichtlineare Kurven im HSV-Modell.*

5.2.2 Darstellungsmodelle

Der RGB-Würfel leitet sich von den mit CRT-darstellbaren Farbkanälen ab. Das CMY(K)-Modell dagegen stammt aus der Foto- bzw. Druckindustrie. Die Umrechnungsformel der Farbvektoren geschieht einfach über

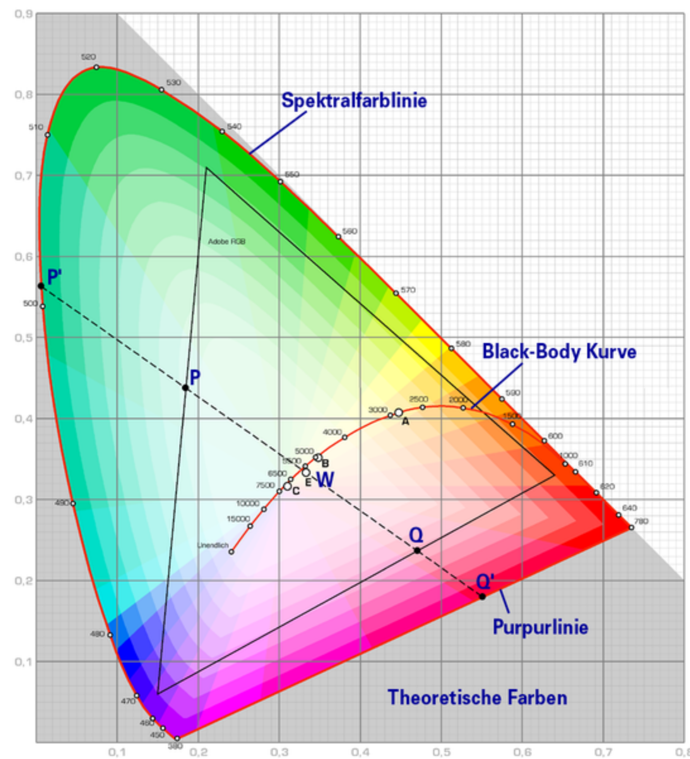


Abbildung 5.9. Farben im CIE-Diagramm

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix} \qquad \left(\text{resp.} \begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix} \right).$$

Das CMYK-Modell sieht zu den Farben noch einen weiteren Kanal Schwarz (K von *Blac**K***) vor. Für den Druckprozess ist das vor allem aus Kostengründen wichtig (farbige Pigmente sind sehr viel teurer als schwarze). Außerdem hängt das Mischen von Farben auf dem Papier von weiteren Faktoren ab. Die Beimischung eines Schwarzkanals erbringt meist die größere Farbtiefe. Nach der Annahme, dass ein neutraler Grauton sich zu gleichen Teilen aus allen drei Farbkanälen zusammensetzt, ist der vierte Kanal dazu vorgesehen, diesen Sockelwert aus dem Minimum aller drei Farbkanäle beizusteuern.

$$\boxed{\begin{aligned} K &= \min(C, M, Y) \\ C_K &= C - K \\ M_K &= M - K \\ Y_K &= Y - K \end{aligned}}$$

Wahrnehmungsbegriff		Erklärung	Physikalischer Begriff
englisch	deutsch		
Hue	Farbwert	rot/gelb/grün/ cyan/blau/magenta	Dominante Wellenlänge
Saturation	Sättigung	Entfernung vom Grauton gleicher Helligkeit	Anregungsgrad
Brightness **L**ightness **V**alue	(emittierte) (reflektierte) Helligkeit		Helligkeit oder Luminanz

Tabelle 5.2. Das HSV-Modell

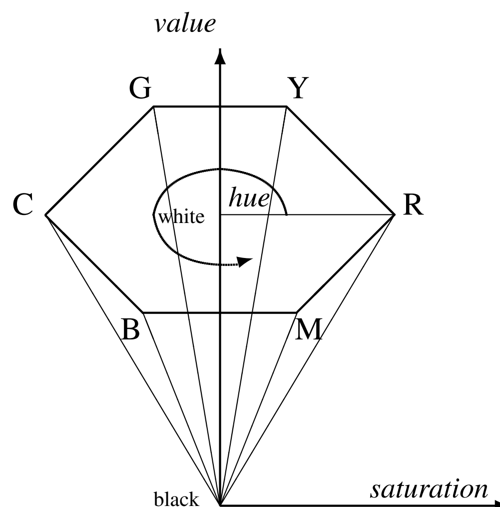


Abbildung 5.10. Das HSV-Hexagon ähnelt in der Anordnung der Farben dem RGB-Würfel.

Addiert man Farben auf einem weißen Trägermaterial, so entsteht beim Überlagern aller Farben schließlich Schwarz. Beim subtraktiven Mischen am Bildschirm wird entsprechend vom weißen Licht nach und nach Farbe abgezogen, daher die Dualität beider Farbmodelle.

Beim additiven Mischen aus den Primärfarben wird dem Hintergrundschwarz jeweils ein weiterer Kanal mit einer der Farben R, G oder B addiert. Schließlich entsteht bei voller Intensität aller Kanäle ein Weiß.

Als CMY- oder RGB-Würfel bezeichnet man die im 3D-Farbraum aufgetragenen Einheitswürfel, deren jeweilige primäre Farbkanäle auf den Achsen aufgetragen werden. In natürlicher Weise ergeben sich die Mischfarben an den entsprechenden Eckpunkten und Weiß resp. Schwarz stellt den Ursrung dar. Entlang der Raumdiagonale befinden sich alle neutralen Grauwerte bis hin zur maximalen Intensität.

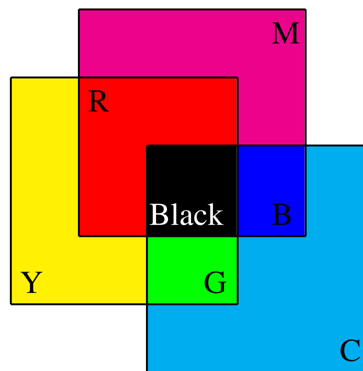


Abbildung 5.11. Das CMY-Modell

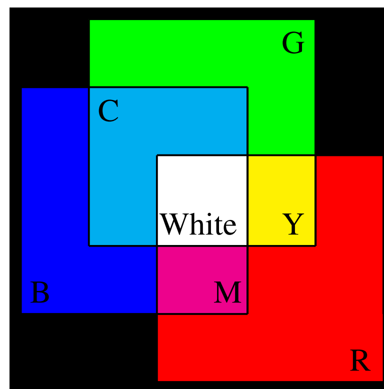


Abbildung 5.12. Das RBG-Modell

5.2.3 Übertragungsmodelle

Der Videotechnik liegen die häufig national fixierten Fernsehnormen zugrunde. Daher unterscheidet man im wesentlichen die amerikanische Norm des kommerziellen US-Fernsehens, das mit 525 Rasterzeilen arbeitet und unter der Abkürzung NTSC (für **N***ational* **T***elevision* **S***ystem* **C***ommittee*) firmiert (Bildschirmauflösung: 646 $\times$ 486 Quadratpixel). Andererseits arbeitet das europäische Fernsehen mit der PAL-Norm auf 625 Rasterzeilen (Bildschirmauflösung: 768 $\times$ 576 Quadratpixel). Die größere Auflösung ist ein Vorteil der späteren Entwicklung des europäischen Fernsehens. In Frankreich ist SECAM eine verbreitete Variante.

Die Luminanz Y nimmt die größte Bandweite ein, denn die menschliche Wahrnehmung kann Helligkeitsunterschiede deutlicher erkennen und lokalisieren, als etwa Farbunterschiede. Ein weiterer Grund ist die (Abwärts-)Kompatibilität mit Schwarz-Weißgeräten. Sie lesen nur die Y-Komponente aus.

Die Chrominanzwerte werden mit geringerer lokaler Auflösung gespeichert, was häufig den Eindruck erweckt, dass die Farbe auf den Objekten zu schwimmen scheint.

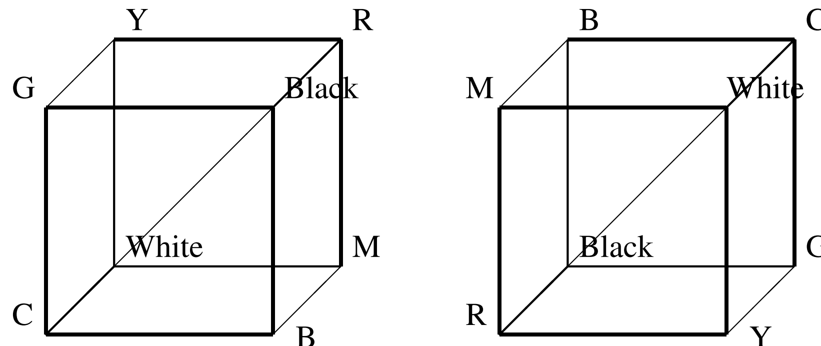


Abbildung 5.13. CMY-Würfel (links) und RBG-Würfel (rechts).

	Bandbeite	Herkunft der Bezeichnung	Name
Y	4.0 MHz	Y aus den CIE-Primitiven	Luminanz
I	1.5 MHz	Modulationsmethoden	Chrominanz
Q	0.6 MHz	zum Kodieren des Trägersignals	Chrominanz
Y	5.5 MHz	Y aus den CIE-Primitiven	Luminanz
U	1.9 MHz	Modulationsmethoden	Chrominanz
V	1.9 MHz	zum Kodieren des Trägersignals	Chrominanz

Tabelle 5.3. Bandweiten der US-amerikanischen und europäischen Videonorm, aus Foley et al. [FvDF+96a]

Um die YIQ-Werte aus den RGB-Werten zu berechnen, gibt es eine simple lineare Matrix, die darauf beruht, dass die Chrominanz in I und Q aus jeweils einer roten und blauen Differenzkomponente zur Luminanz besteht.

$$\begin{array}{rcccccl} I & = & 0.74(R-Y) & - & 0.27(B-Y) & & \\ Q & = & 0.48(R-Y) & + & 0.41(B-Y) & & \\ & & \text{rote} & & \text{blaue} & & \text{Differenzkomponente} \end{array}$$

Gerundet auf die zweite Nachkommastelle ist die folgende Matrixmultiplikation äquivalent zu den obigen Gleichungen:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.528 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

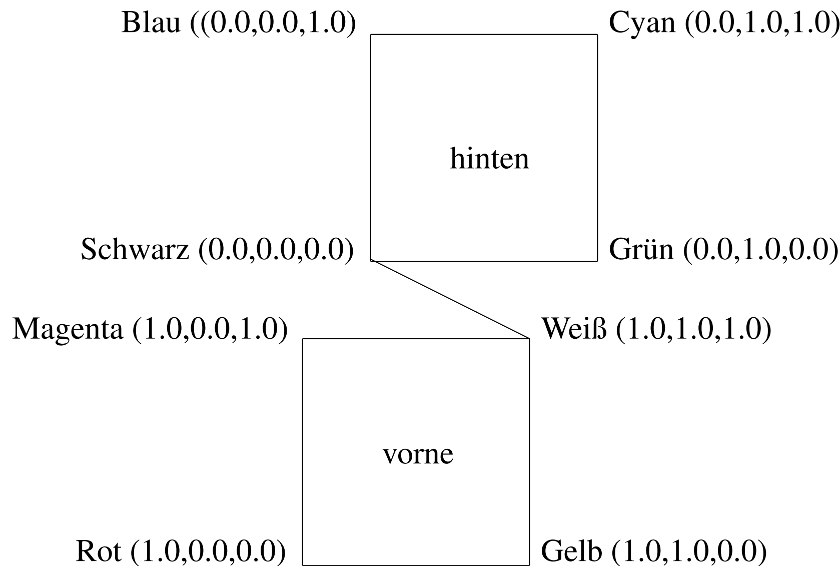


Abbildung 5.14. Die Farbwerte des RBG-Würfels.

Die inverse Berechnung ergibt sich aus:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.956 & 0.621 \\ 1 & -0.272 & -0.647 \\ 1 & -1.105 & 1.702 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

Das Luminanzsignal Y ist in beiden Standards gleich, jedoch hat man sich später für das getrennte Kodieren von blauer und roter Differenzkomponente entschieden. Damit wird ein so genanntes *Chromacrosstalk* vermieden. Das wird an der Transformationsmatrix (bzw. ihrer Inversen) sichtbar.

$$\left.\begin{array}{rcll} U & = & 0.493(B-Y) & \text{blaue} \\ V & = & 0.877(R-Y) & \text{rote} \end{array}\right\} \text{Differenzkomponente}$$

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.437 \\ 0.615 & -0.515 & -0.1 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Die inverse Berechnung ergibt sich aus:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.14 \\ 1 & -0.394 & -0.581 \\ 1 & 2.028 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix}$$

Bei räumlicher Modulationsfrequenz von 4:2:2 und einer empfohlenen 8 bit-Kodierung von Y erreicht man immerhin eine Farbtiefe von ungefähr 16 bit gegenüber einer 24 bit RGB Farbtiefe bei einem Standard-Computerschirm. Das berücksichtigt noch immer nicht die viel größere Auflösung eines solchen Schirms und seine wesentlich höhere Bildwiederholfrequenz, alles Vorzüge einer nochmals späteren Technik.

5.3 Farbinterpolation und Schattierung

Bei der Initialisierung eines Graphikprogramms legt man sich zu Anfang auf ein Farbmodell fest. Mit der *OpenGL*-Erweiterungsbibliothek *GLUT* geschieht das bereits über einen Aufruf der Routine `glutInitDisplayMode(...|GLUT_RGB|...);` Wahlweise kann auch `GLUT_RGBA` aufgerufen werden. Farben werden dann entweder mit dem Befehl

```
glColor3f(GL_FLOAT r, GL_FLOAT g, GL_FLOAT b);
```

angegeben, wobei die drei Floatvariablen die Intensität der jeweiligen Farbkanäle rot, grün und blau zwischen Null und 1.0 angeben. Oder der Befehl enthält noch eine vierte Alphakomponente, die die Transparenz der Farbe betrifft.

```
glColor4f(GL_FLOAT r, GL_FLOAT g, GL_FLOAT b, GL_FLOAT α);
```

`glColor4fv(...);` erwartet einen Vektor der Länge vier. Die Farbe ist eine Zustandsvariable, die solange aktiv bleibt, bis sie durch einen erneuten Farbaufruf geändert wird.

5.3.1 Gouraud Shading

Abbildung 5.15. Henri Gouraud (1944*)

Das auf Henri Gouraud (1971) zurückgehende Verfahren meint eine lineare Farbinterpolation entlang der *Scanline*. Der Farb- (oder Intensitäts-)Wert I_s an dem Punkt P_s eines Polygons wird dabei nach einer Formel bestimmt, die zunächst entlang der Kante aus den Farben der Eckpunkte linear interpolierte Intensitäten bestimmt und dann diese entlang der Scanline interpoliert.

Sei I_j die Intensität am Punkt $P_j = (x_j, y_j)$, dann gilt

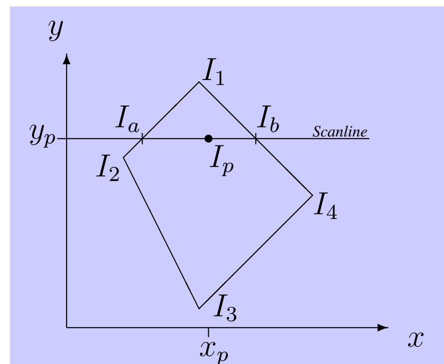


Abbildung 5.16. Intensitätsberechnung entlang der *Scanline*

$$I_a = I_1 - (I_1 - I_2)\,\frac{y_1 - y_p}{y_1 - y_2}$$

$$I_b = I_1 - (I_1 - I_4)\,\frac{y_1 - y_p}{y_1 - y_4}$$

$$\Rightarrow \quad \boxed{I_p = I_a - (I_a - I_b)\,\frac{x_a - x_p}{x_a - x_b}}$$

Die inkrementelle Implementierung verkürzt dieses Verfahren, indem entlang der *Scanline* das Intensitätsinkrement berechnet wird.

$$\Delta I_p = \frac{\Delta x}{x_a - x_b}(I_a - I_b); \qquad I_{p,n+1} = I_{p,n} + \Delta I_p$$

Bekannte Probleme des *Gouraud Shading* sind Anomalien beim Drehen, da die *Scanline* das Polygon in Bildschirm- und nicht in Weltkoordinaten schneidet. Seltsame Effekte bei Anwendung von Lichtmodellen behebt man am besten durch die ausschließliche Anwendung dieses *Shading* auf die diffuse Komponente.

Exkurs: *OpenGL-Implementierung*

Gouraud Shading ist standardmäßig im *OpenGL* RGB(A)-Modus enthalten: Jeder Knoten zwischen `glBegin();` und `glEnd();` kann eine neue Farbe erhalten, wobei automatisch mit der Voreinstellung für `glShadeModel(GL_SMOOTH);` das Gouraud Shading, also die lineare Farbinterpolation implementiert ist. Dagegen färbt `glShadeModel(GL_FLAT);` jedes Polygon ausdrücklich einfarbig. Dabei wird

bei einfachen Polygonen die Farbe des ersten Knotens genommen, während bei komplizierteren Polygonpflasterungen *(Tesselations)* der letzte Knoten die Farbe entscheidet.

Bemerkung 5.3 *Farbe kann generell und das Wechseln der Shading-Modelle kann besonders gut dazu eingesetzt werden, Fehler beim Indizieren von Punkten zu finden.*

5.3.2 Phong Shading

Das von Bui-Tuong Phong 1975 vorgeschlagene und nach ihm benannte *Phong Shading* ist eine Weiterentwicklung des *Gouraud Shading*, das nicht nur das Streulicht (bi-)linear interpoliert, sondern auch das spiegelnd reflektierte Licht richtig handhabt.

Statt Intensitäten werden Normalen interpoliert.

Man berechnet die Normale N eines Dreiecks aus den drei Eckpunkten P_1, P_2 und P_3 über ein Vektor-Kreuzprodukt $\tilde{N} = (P_1 - P_2) \times (P_2 - P_3) = v \times w$. Das Kreuzprodukt in drei Dimensionen ist definiert über

$$\tilde{N} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \times \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} v_2 w_3 - v_3 w_2 \\ v_3 w_1 - v_1 w_3 \\ v_1 w_2 - v_2 w_1 \end{pmatrix} = \begin{pmatrix} \tilde{n}_1 \\ \tilde{n}_2 \\ \tilde{n}_3 \end{pmatrix}.$$

Anschließend wird der Vektor $\tilde{N}$ komponentenweise durch seine euklidische Norm $\sqrt{(\tilde{n}_1^2 + \tilde{n}_2^2 + \tilde{n}_3^2)}$ dividiert, also auf eine Länge von Eins normiert.

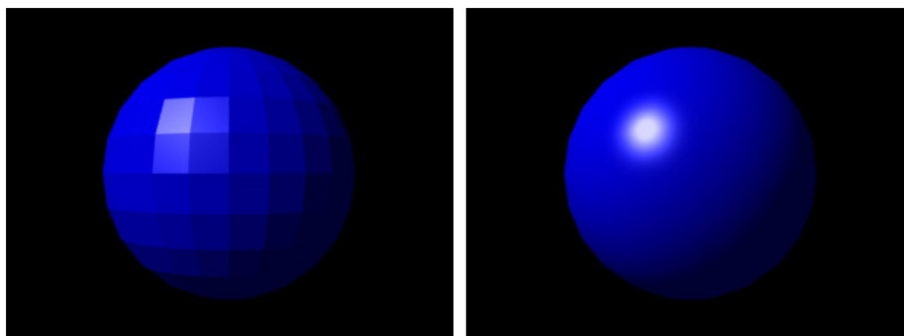


Abbildung 5.17. Links sieht man eine aus ca. 200 Punkten erstellte Kugel mit Flat Shading, rechts die Kugel aus den selben Punkten mit Phong Shading.

Mit drei Punkten ist bereits eine Ebene festgelegt. Durch die Reihenfolge, in der diese Punkte durchlaufen werden, legt man eine Orientierung dieser Ebene fest, wobei eine Laufrichtung gegen den

Uhrzeigersinn bedeutet, dass die Ebene uns ihre Vorderseite zeigt. Tatsächlich wird die Normale einem Eckpunkt zugeordnet und bleibt solange gültig, bis erneut einem Eckpunkt eine andere Normale zugewiesen wird. Daher ist es sinnvoll, die Normale in einem Eckpunkt über die Flächennormalen aller an diesen Eckpunkt grenzenden Dreiecke geeignet zu mitteln.

Die Normalen, die jedem einzelnen Eckpunkt zugeordnet sind und in diesen Punkten senkrecht zu einer Tangentialebene an das Objekt ausgerichtet sein sollen, unterscheiden sich jetzt deutlich von einer zentrierten Flächennormalen eines einzelnen (ebenen) Polygons. Das Interpolieren der Normalen ist daher der natürliche Zugang zur Darstellung gerundeter Objekte, bei denen der Aufbau aus planaren Dreiecken letztlich nur ein Hilfsmittel ist, mit dem man die Gestalt komplizierter dreidimensionaler Körper aus einer relativ geringen Anzahl von Eckpunkten realisieren kann.

Bemerkung 5.4 *Die Normalen einer Kugel sind identisch mit den (normierten) Punkten auf der Kugeloberfläche, da diese Vektoren automatisch senkrecht zur Kugeloberfläche ausgerichtet sind.*

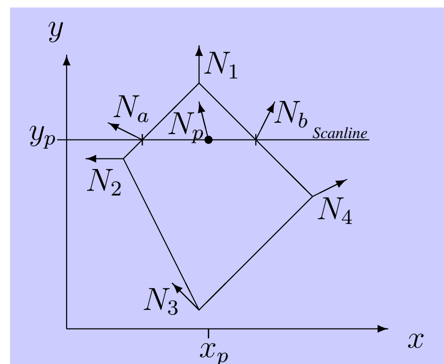


Abbildung 5.18. Die Normalen werden entlang der *Scanline* interpoliert.

Sei N_j die Normale im Punkt $P_j = (x_j, y_j)$, dann gilt

$$N_a = N_1 - (N_1 - N_2)\,\frac{y_1 - y_p}{y_1 - y_2}$$

$$N_b = N_1 - (N_1 - N_4)\,\frac{y_1 - y_p}{y_1 - y_4}$$

$$\Rightarrow \quad \boxed{N_p = N_a - (N_a - N_b)\,\frac{x_a - x_p}{x_a - x_b}}$$

Hier handelt es sich allerdings um Vektorgleichungen, d. h. gegenüber dem *Gouraud Shading* muss ein mindestens ***dreifacher Aufwand*** betrieben werden: Die drei Raumrichtungen müssen einzeln interpoliert und korrekterweise muss der so gewonnene Vektor anschließend wieder normiert werden. Die inkrementelle Implementierung verkürzt auch hier das Verfahren, indem entlang der *Scanline* das Normaleninkrement, jetzt aber für jede Raumrichtung einzeln berechnet wird.

$$\Delta N_{\substack{p,x\\p,y\\p,z}} = \frac{\Delta x}{x_a - x_b}\left(N_{\substack{a,x\\a,y\\a,z}} - N_{\substack{b,x\\b,y\\b,z}}\right); \qquad N_{\substack{p,x,n+1\\p,y,n+1\\p,z,n+1}} = N_{\substack{p,x,n\\p,y,n\\p,z,n}} + \Delta N_{\substack{p,x\\p,y\\p,z}}$$

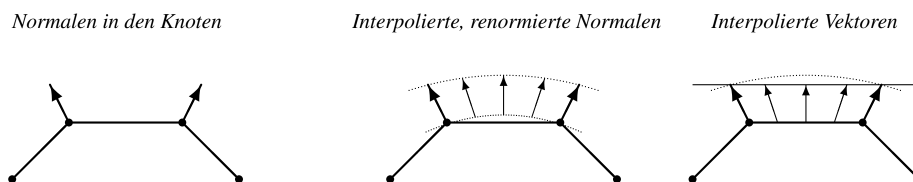


Abbildung 5.19. Das Resultat der beiden Interpolationsmethoden im Vergleich, links *Gouraud Shading*, rechts *Phong Shading*, normiert oder einfach interpoliert.

Ein beschleunigtes Verfahren verzichtet einfach auf das Renormieren der interpolierten Vektoren. Der dabei auftretende Fehler ist eher gering, vor allem je dichter die Punkte zur Darstellung des gekrümmten Objekts gewählt sind.

Eine andere Methode ist der so genannte H-Test, wobei nach *Highlights* gesucht wird. Diese werden durch *Gouraud-Shading* unter Umständen gar nicht wiedergegeben, stellen aber möglicherweise nur einen geringen Flächenanteil. Dann beschränkt man das *Phong-Shading* auf die Bereiche mit *Highlights*, was allerdings nur lohnt, wenn der Suchaufwand für einzelne Highlights deutlich unter dem Aufwand für das *Phong-Shading* der gesamten Szene liegt.

5.4 Phong Modell, ein einfaches Reflexionsmodell

Nachdem im vorigen Abschnitt über *Phong-Shading* bereits auf die Bedeutung von äußeren Flächennormalen hingewiesen wurde, soll hier das einfache *Phong-Modell* erklärt werden. Die Farbe eines Pixels wird dabei über seine Zugehörigkeit zu einem Objekt mit gewissen Materialeigenschaften und über die Farben und Positionen der Lichtquellen definiert.

Das Modell greift dabei auf die Lambertsche Regel zurück und definiert Komponenten aus Umgebungslicht, das mit *Ambient* bezeichnet wird, diffusem *Streulicht* und spiegelndem *Spekularlicht*.

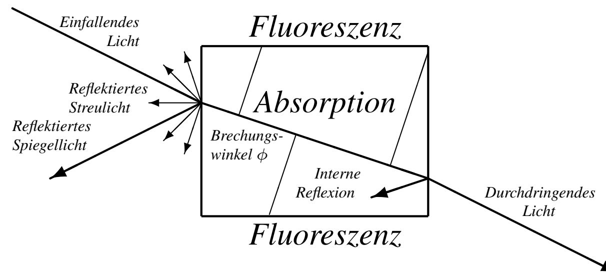


Abbildung 5.20. Licht trifft auf einen Körper.

$$I = I_{ambient} + I_{diffus} + I_{specular}$$

Da es meist keinen perfekt matten, einzig diffus streuenden Körper gibt, und da auch sein Gegenteil, der perfekt spiegelnde, reflektierende Körper eigentlich nicht real ist, werden die genannten Komponenten geeignet gewichtet und anschließend addiert. Die Intensität der drei Farbkanäle an einem Punkt der Oberfläche des Objekts ist schließlich eine Linearkombination aus diesen drei Farbintensitäten.

5.4.1 Umgebungslicht

Das Umgebungslicht wird einfach als ein konstanter Term $I_a k_a$ addiert und soll das Steulicht an allen möglichen umgebenden Objekten modellieren. Diesen Term präziser zu berechnen, muss erheblicher Aufwand getrieben werden, wobei dann allerdings auch sehr realistische Bilder erzeugt werden können (siehe das Kapitel über *Radiosity*). Wenn im *Direct Rendering* aber einfach ein Lichtmodell aktiviert ist, aber noch keine Lichtquellen positioniert oder Oberflächennormalen bekannt sind, so erscheinen die Objekte alle in einem voreingestellten ambienten Licht. Als Formel kann dies über

$$I = I_{ambient} = I_a k_a$$

ausgedrückt werden, wobei I_a die Intensität des Umgebungslichts und k_a eine Materialkonstante ist.

5.4.2 Diffuses Streulicht

Ein grünes Objekt erscheint grün, weil es aus dem durch die Lichtquelle angebotenen Spektrum ***alle*** bis auf die grüne Komponente absorbiert und ***nur*** grün reflektiert. Hier wird erstmalig die Bedeutung der Normalen eines Objekts klar, denn aus dem Winkel θ zwischen der Lichtquelle L und der Oberflächennormalen N berechnet sich die Intensität des Streulichts. Ein perfekt streuendes Objekt streut gleichmäßig in alle Richtungen. Der Eindruck beim Betrachter ist daher von seiner Position V ***unabhängig***.

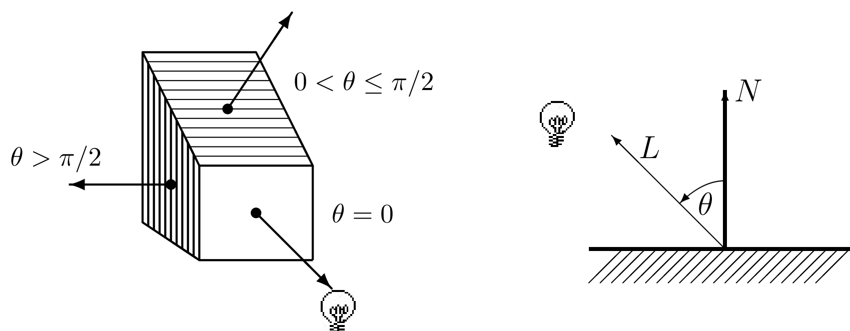


Abbildung 5.21. Das diffuse Streulicht hängt einzig vom Einfallswinkel des Lichts ab, aber nicht von der Betrachterposition.

Im Folgenden sind die Vektoren L, N und V als normierrt angenommen, d.h. ihre Länge ist 1. Daher kann der $\cos\theta = L \cdot N$ als Skalarprodukt geschrieben werden.

$$\begin{aligned} I_{diffus} &= I_i k_d \cos\theta \qquad 0 \leq \theta \leq \pi/2 \\ &= I_i k_d (L \cdot N) \end{aligned}$$

Hierbei ist I_i die Intensität des einfallenden Lichts und k_d eine weitere Materialkonstante, ein Reflexionsfaktor für die diffuse Komponente des Materials. Bei n Lichtquellen ist die Summe über alle einzelnen Intensitätsquellen und die Richtungen zu bilden.

$$I_{diffus} = k_d \sum_n I_{i,n}(L_n \cdot N)$$

5.4.3 Spiegelnde Reflexion

Die spiegelnde Reflexion stellt sich etwas komplexer dar. Hier wird die Normale eines Objekts benutzt, um die Reflexionsrichtung zu bestimmen, denn ein spiegelnder Körper reflektiert das einfallende Licht nach der simplen Regel:

Einfallswinkel = Ausfallswinkel

Ein perfekt spiegelnder Körper, der von einer punktförmigen Lichtquelle[2] beleuchtet wird, erscheint aus einer ***einzigen Richtung*** betrachtet in genau der Farbe der Lichtquelle hell, sonst ist er dunkel.

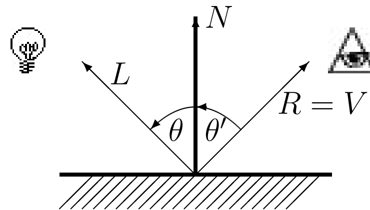


Abbildung 5.22. Bei einem perfekt spiegelnden Körper wird die punktförmige Lichtquelle ausschließlich in einer einzigen Richtung wirksam, so dass der Betrachter genau aus dieser Richtung auf das Objekt schauen muss, um die Wirkung des Lichts zu sehen.

$$I_{specular} = I_i k_s \cos^n \phi = I_i k_s (R \cdot V)^n$$

Das einfache Reflexionsmodell von Phong benutzt den Winkel ϕ zwischen den Vektoren R und V (siehe Abbildung 5.23) und wichtet mit dem Exponenten n, wieviel reflektiertes Licht der Betrachter tatsächlich zu sehen bekommt. Hierbei ist I_i wieder die Intensität des einfallenden Lichts und k_s eine weitere Materialkonstante, ein Reflexionsfaktor für die spiegelnde Komponente des Materials.

In der Summe aus ambienter, diffuser und spekularer Komponente ergibt sich nun die vollständige Formel für das Phong Modell:

$$I = I_a k_a + I_i [k_d (L \cdot N) + k_s (R \cdot V)^n]$$

Im rechten Teil der Abbildung 5.24 ist die Summe aus den Einzelkomponenten farblich dargestellt: der Funktionswert des diffusen Anteils ist grün gekennzeichnet. Dazu addiert sich die spiegelnde Reflexion, die für verschiedene Exponenten in rot, mangenta oder blau je nach Größe des Exponenten n dargestellt ist. Ein perfekt spiegelndes Objekt wichtet die Reflexion mit einem Exponenten $n = \infty$.

Eine Gegenüberstellung von diffuser und spiegelnder Reflexion soll nochmal betonen, dass die diffuse Komponente von der Betrachterposition unabhängig ist, dafür aber hier die Farbe des reflektierten Lichts aus der Materialeigenschaft und der Eigenschaft der Lichtquelle berechnet wird. Dagegen berücksichtigt die betrachterabhängige spiegelnde Reflexion (Highlight) nur die Farbe der Lichtquelle.

[2] Man denke an einen Laserpointer.

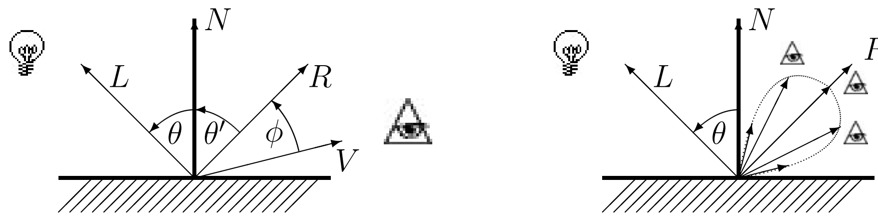


Abbildung 5.23. Spiegelnde Reflexion, rechts mit einer Wichtungskurve um den Vektor R für verschiedene Betrachterpositionen.

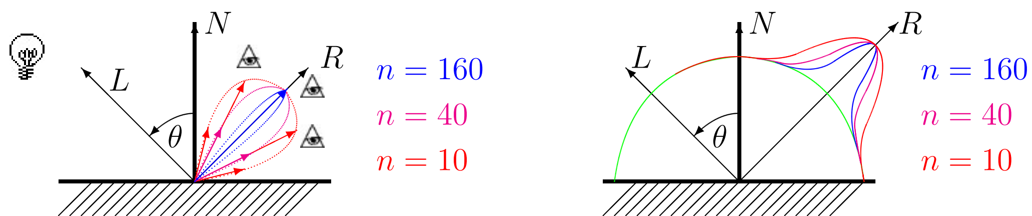
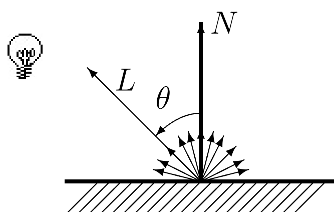
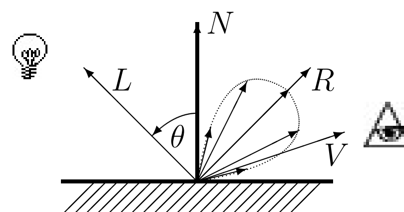


Abbildung 5.24. Die roten, magenta-farbenen und blauen Vektoren zeigen in Richtung von Betrachterpositionen V. Sie markieren den Spekularanteil der Gesamtintensität und hängen in der Länge von $\cos^n \phi = (R \cdot V)^n$, also dem in gleicher Farbe dargestellten Exponenten n ab. Rechts sind diese Werte auf die grüne Kurve für die diffuse Komponente addiert.

Diffuse Reflexion	**Spiegelnde Reflexion**
unabhängig von der Betrachterposition	**abhängig** von der Betrachterposition
Farbe des reflektierten Lichts = Farbe des **Objekts** + Farbe der Lichtquelle	Farbe des reflektierten Lichts = Farbe der Lichtquelle





5.4.4 Blinn-Beleuchtungsmodell, eine Vereinfachung des Phong Modells

James F. Blinn (Jim Blinn) hat 1977 das Phong Modell vereinfacht: Das aufwendige Berechnen von R wird durch eine Modellannahme erheblich reduziert: Die Lichtquelle befindet sich bei Unendlich, wodurch parallel einfallendes Licht erzeugt wird. Damit ist $N \cdot L$ für eine gegebene Fläche konstant. Der Betrachter befindet sich an fester Position. Statt aufwendig aus L und N den Vektor $R = 2N(N \cdot L) - L$ zu berechnen, ermittelt man jetzt einen Vektor H, der die Normale zu einer hypothetischen Oberfläche eines perfekt spiegelnden Körpers in Betrachterrichtung darstellt. Diese Berechnung ist einfach $H = (L+V)/||L+V||$. Der spiegelnde Anteil wird jetzt aus dem Skalarprodukt $(N \cdot H) = \cos\varphi$ berechnet. Der Vektor H wird auch als Halfway-Vektor bezeichnet.

Abbildung 5.25. Jim Blinn

Der Betrag des Winkels zwischen der tatsächlichen Normalen N und der hypothetischen Normalen H ist nur halb so groß wie der Winkel ϕ zwischen der Reflexionsrichtung R und der Betrachterposition V. Das ergibt sich simpel aus der Konstruktion, da die Abweichung des Betrachters von der perfekt spiegelnden Richtung R je zur Hälfte dem Einfallswinkel und dem Ausfallswinkel des Lichts zugeordnet wird. Dieser Fehler kann aber mit einem größeren Exponenten m kompensiert werden.

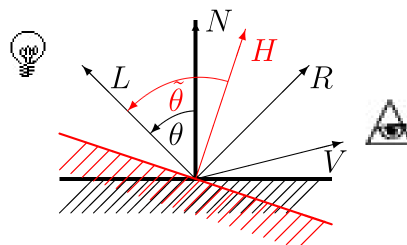


Abbildung 5.26. Vereinfachungen der spiegelnden Reflexion.

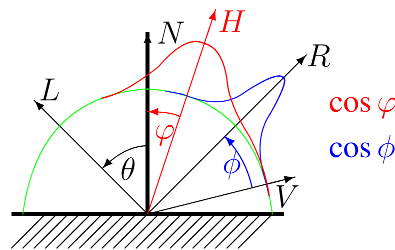


Abbildung 5.27. Der kleinere Winkel φ erzeugt ein größeres Highlight, das außerdem gegenüber der korrekten Position verschoben dargestellt wird.

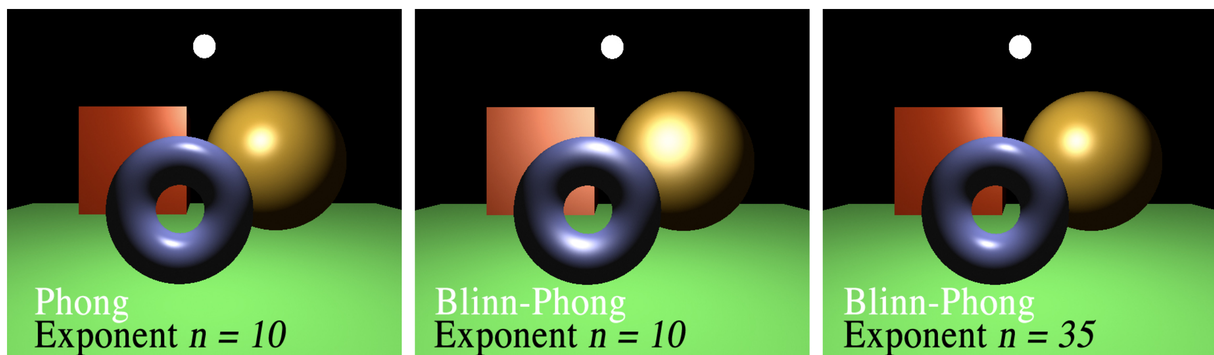


Abbildung 5.28. Links sieht man das ursprüngliche Phong Modell, in der Mitte das mit dem Halfway-Vektor vereinfachte Blinn-Phong Modell und rechts das Blinn-Phong Modell mit größerem Exponenten.

Die vereinfachte Formel lautet nun:

$$\boxed{I = I_a k_a + I_i[k_d(L \cdot N) + k_s(N \cdot H)^n]}$$

wobei natürlich auch hier gilt, dass bei mehreren Lichtquellen die Summe über alle Intensitäten und Positionen der verschiedenen Quellen gebildet werden muss.

$$I = I_a k_a + k_d \sum_j I_{i,j}(L_j \cdot N) + k_s \sum_j I_{i,j}(N \cdot H_j)^n$$

Bemerkung 5.5 *Bei konstanter Lichtposition L (und festem Betrachter V) ist die Intensität nur noch eine Funktion der Flächennormale, bei ebenen Polygonen ohne Normalenänderung also konstant über das ganze Polygon. Bewegt man den Betrachter, variieren auch* Highlights *geringer, wenn L konstant ist.*

Bemerkung 5.6 *Ein einfaches Farbmodell basiert auf den nach Farbkanälen aufgespaltenen Intensitäten $I = I_r + I_g + I_b$ mit*

$$\begin{aligned} I_r &= I_a k_{a_r} + I_i[k_{d_r}(L \cdot N) + k_s(N \cdot H)^n] \\ I_g &= I_a k_{a_g} + I_i[k_{d_g}(L \cdot N) + k_s(N \cdot H)^n] \\ I_b &= I_a k_{a_b} + I_i[k_{d_b}(L \cdot N) + k_s(N \cdot H)^n] \end{aligned}$$

5.4.5 Abschwächung des Lichts durch Abstände

Eine einfache Verbesserung erzielt man durch die Modellierung der Streuung des Lichts aus der dargestellten Szene an Dunst, Schwebteilchen oder einfach Luft, die von der Entfernung des Objekts zum Betrachter abhängig ist. In der Summe stellt sich die Intensität jetzt aus der ambienten und den reziprok mit der Entfernung gewichteten diffusen und spiegelnden Komponenten dar:

$$\boxed{I = I_a k_a + I_i[k_d (L \cdot N) + k_s(N \cdot H)^n]/(r + k)}$$

Der Abstand zum Betrachter r wird mit einer Konstanten $k > 0$ jedenfalls auf einen positiven Wert gesetzt, um bei dieser einfachen Modellierung die Division durch Null zu vermeiden. Diese lineare Funktion kann aber auch durch exponentielles Abfallen der Intensität ersetzt werden, wie es beispielsweise beim so genannten *Depth-cueing* mit den *fog equations* geschieht.

OpenGL-Exkurs über *Fog* und *Depth-cueing*

Nebel oder Dunst *(Fog)* streuen das von einem Objekt ausgehende Licht auf dem Weg zum Betrachter. Innerhalb von *OpenGL* stehen verschiedene Funktionen zur Verfügung, um einen Tiefeneindruck zu erzeugen, der auf der Entfernung des Betrachters vom Objekt beruht.

$$\begin{aligned} \texttt{GL_EXP} &\longrightarrow & f &= e^{-density \cdot z} \\ \texttt{GL_EXP2} &\longrightarrow & f &= e^{-density \cdot z^2} \\ \texttt{GL_LNEAR} &\longrightarrow & f &= \frac{z_{end} - z}{z_{end} - z_{start}} \end{aligned}$$

Diese Funktionen werden über die folgenden Kommandos aufgerufen:

```
glEnable(GL_FOG);
glHint(GL_FOG_HINT, GL_DONT_CARE);
glFogi(GL_FOG_MODE,GL_EXP);
glFogfv(GL_FOG_COLOR, GLfloat[4] fogColor);
glFogf(GL_FOG_DENSITY, GLfloat density);
glFogf(GL_FOG_START, GLfloat start);
glFogf(GL_FOG_END, GLfloat end);
```

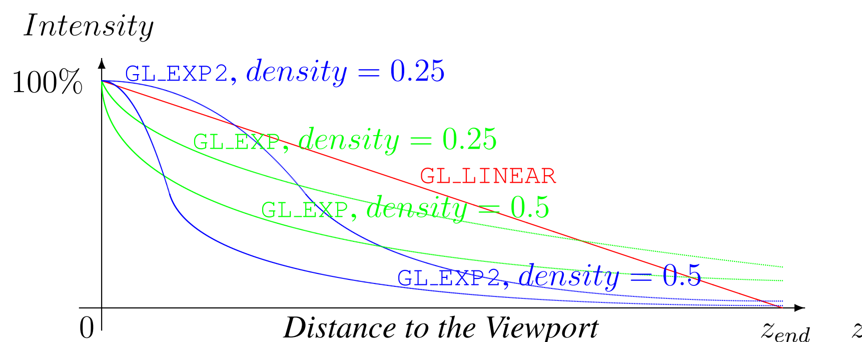


Abbildung 5.29. Funktionen zur Modellierung der atmosphärischen Streuung.

Bisher werden zwei parallele Flächen identischen Materials, die aus der gleichen Richtung beleuchtet werden, aber unterschiedlichen Abstand zur Lichtquelle haben, in identischer Farbe gezeichnet. Auch für die Lichtquellen gilt, dass für die Intensität des reflektierten Lichts die Entfernung der Quelle zum Objekt berücksichtigen kann. Das geschieht mit einem Abschwächungsfaktor (*Attenuation*) $f_{att} = f_{att}(d_L)$, wobei d_L der Abstand zur Lichtquelle ist. Typischerweise ist

$$f_{att} = \max\left(\frac{1}{c_1 + c_2 d_L + c_3 {d_L}^2} \text{ , } 1 \right).$$

Dabei sorgt der konstante Term c_1 dafür, dass dieser Faktor nicht unendlich groß wird, während mit den linearen und den quadratischen Termen für weit entfernte und näher gelegene Objekte eine Variation der Lichtstärke modelliert werden kann.

5.4.6 Grenzen der einfachen Reflexionsmodelle

Einfache Reflexionsmodelle sind lokale Modelle und berücksichtigen keine Verdeckungen der Lichtquelle durch andere Objekte. Es wird ausschließlich ein Farbwert an einer bestimmten Stelle eines Objekts berechnet, der sich aus den Vektoren L, N und V und den Material- und Lichtparametern zusammensetzt. Verschattungen werden nicht berücksichtigt. Schlagschatten sind bespielsweise nur über weitere Anstrengungen zu erzielen und nicht im Modellansatz enthalten.

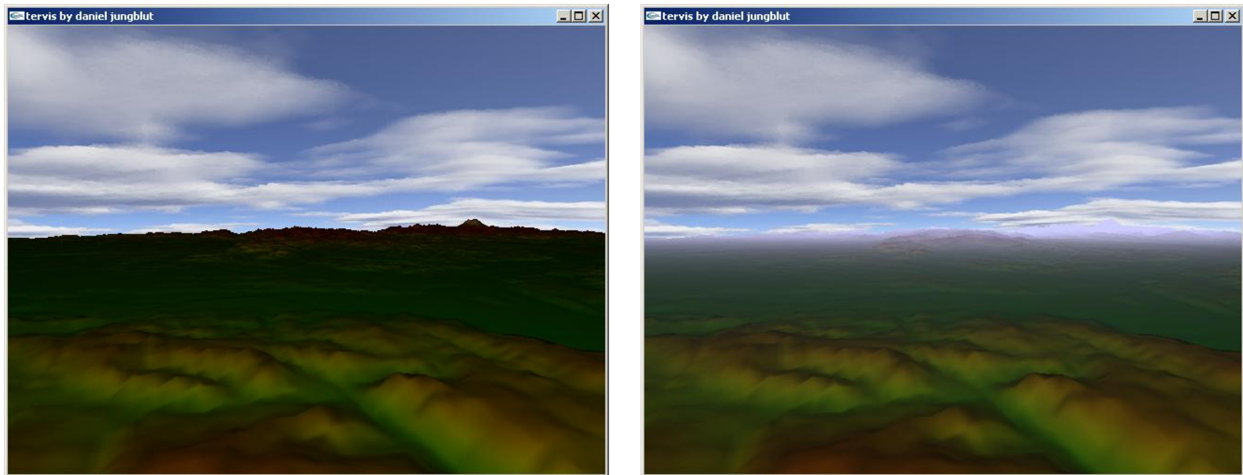


Abbildung 5.30. Links sieht man ein Terrainmodell ohne Fog, rechts mit Fog, wobei hier eine lineare Abschwächung gewählt wurde. Quelle: Daniel Jungblut, IWR, Universität Heidelberg.

Damit ähneln die lokalen Lichtmodelle in etwa dem z-Buffer Algorithmus, der bei transparenten Objekten an seine Grenzen kommt. In beiden Fällen können nur dann bessere Ergebnisse erzielt werden, wenn man zu globalen Modellen wechselt, die im Objektraum arbeiten. Für Raytracing oder Radiosity Modelle ist der Objektraum unerlässlich, denn alle Objekte beinflussen den Lauf der Lichtstrahlen bzw. das energetische Gleichgewicht zwischen allen Flächen im Raum. Wenn man auf alle Objekte und ihre Lage zueinander jederzeit und bei jeder Drehung zugreifen können muss, gibt man die Geschwindigkeitsvorteile zugunsten der Qualitätssteigerung auf.

5.5 Lichtquellen

5.5.1 Headlight

Mit *Headlight* meint man eine Lichtquelle, die direkt aus der Betrachterposition eine Szene beleuchtet. Man denke dabei an eine Grubenlampe, die man sich im Bergbau oder als Höhlenforscher an die eigene Stirn montiert. Hier gilt $L = V$. Allerdings ist die vierte Koordinate der homogenen Koordinaten $w = 0$, wodurch das Licht parallel einfällt und $N \cdot H$ konstant ist. Für viele Programme ist eine derartige Lichtquelle die Standardeinstellung, so auch für *Virtual Reality Markup Language* (VRML) Viewer.

5.5.2 Punktförmige Lichtquellen

Punktförmige Lichtquellen senden radialsymmetrisch gleichmäßiges Licht aus. Wenn die vierte Koordinate $w = 0$ gilt, so fällt auch ihr Licht aus einer bestimmten Richtung parallel ein.

5.5.3 Gerichtete Lichtquellen

Gerichtete Lichtquellen wurden von Warn 1983 eingeführt. Als sogenanntes Warn Modell bezeichnet man ein mit geringem Rechenaufwand erweitertes Phong Modell, bei dem die Quellenintensität zusätzlich vom Winkel ϑ zwischen den Vektoren L_N und $(-L)$ abhängt. Der $\cos\vartheta$ kann wieder über ein Skalarprodukt ausgedrückt werden, mit dem die Intensität gewichtet wird. Dabei sorgt der Exponent s für ein Fokussieren des Lichts: Je größer der Exponent, desto konzentrierter fällt das Licht nur in die ausgezeichnete Richtung L_N.

$$I = I_a k_a + I_i (L_N \cdot (-L))^s \, [k_d \, (L \cdot N) + k_s \, (N \cdot H)^n]$$

Der konische Spot ist eine gerichtete Lichtquelle, bei der ein maximaler Winkel δ angegeben wird. Außerhalb dieses Winkels wird diese Lichtquelle nicht berücksichtigt.

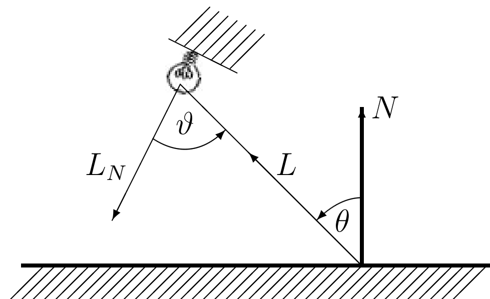


Abbildung 5.31. Quellenintensität hängt von der Richtung ab, in die die Lichtquelle strahlt.

5.6 Zusammenspiel von Licht und Material in OpenGL

In der Graphikbibliothek OpenGL ist ein einfaches Reflexionsmodell voreingestellt. Dieses Modell kann mit `glLightModel(···);` unterschiedliche Algorithmen ansteuern, die aber alle rein lokale Modelle bleiben und daher dem hier vorgestellten Phong Modell nah verwandt sind.

Grundsätzlich gilt, dass mit `glEnable(GL_LIGHTING);` ein Lichtmodell aktiviert wird. Gleichzeitig wird der Befehl `glColor..(···);` außer Kraft gesetzt, denn zunächst ist nicht klar, ob ein Farbwert

der Lichtquelle oder dem Material zuzuschreiben ist. Das Einschalten des Lichtmodells ist vergleichbar mit der Inbetriebnahme einer künstlichen, elektrischen Beleuchtung in einer stockfinsteren Behausung durch Umlegen der Hauptsicherung im Sicherungskasten. Unsere zuvor farbig erdachte, objektweise angestrichene Welt ist zunächst nur notbeleuchtet. Diese Notbeleuchtung besteht aus dem voreingestellten grauen ambienten Licht, das (nachts sind alle Katzen grau) auf ein diffuses, graues Material trifft, das den Objekten zugeordnet wird. Schemenhaft ist die Szene zu erkennen.

Ambientes Licht braucht noch keine Normalen. Jede weitere Komponente einfacher Reflexionsmodelle aber ist dringend auf Normalen angewiesen. Daher sollte spätestens jetzt dafür gesorgt werden, dass wenigstens eine Normale bekannt ist. Dazu stellt man `glNormal..(···);` einem Aufruf von `glVertex..(···);` voran. Die diesem Punkt zugeordnete Normale wirkt sich auch auf alle weiteren Punkte solange aus, bis eine andere Normale zugeordnet wird.

Über bereits voreingestellte oder über vorzugebende Parameter werden Konstanten an das Modell übergeben. Dies geschieht für das Material mit dem Aufruf `glMaterial..(···);`, der an die Ober- oder Unterseite (oder an beide Seiten) der nachfolgend zu zeichnenden Polygone für eine zu benennende Komponente des Lichtmodells Parameterwerte richtet. Dies können diffuse oder spiegelnde Farbwerte oder Exponenten für die Größe der Highlights sein. Auch die ambienten Farbwerte des Materials können nachjustiert werden.

Die Lichtquellen erhalten ihre Farbeinstellungen über `glLight..(···);`, wobei jede Implementierung wenigstens acht Lichtquellen hat, die einzeln für die jeweiligen Komponenten mit Parametern versorgt werden können. Außer ambienten, diffusen und spekularen Farbwerten sind hier die Position, die Spotlight-Eigenschaften und die Art der Abnahme der Intensität mit der Entfernung der Lichtquelle vom Objekt zu regeln. Über das beschriebene Phongmodell hinaus kann ein Licht unterschiedlich farbig mit der diffusen und der spekularen Komponente verrechnet werden. Außerdem kann auch der Intensitätsabfall mit dem Abstand der Lichtquelle zum Objekt modelliert werden.

Erst jetzt hat es Sinn, diese einzelnen Lichtquellen einzuschalten, was mit `glEnable(GL_LIGHT0);` oder `glEnable(GL_LIGHTi);` für die i-te Lichtquelle geschieht.

Legt man mit `glDisable(GL_LIGHTING);` die Hauptsicherung erneut um, erscheint wieder die mit `glColor..(···);` angestrichene Spielzeugwelt ohne Lichtmodell. Unsere Erwartungshaltung war allerdings sicher eine ganz andere: Mit Einschalten des Lichts sollte die bunte Spielzeugwelt zusätzlich das Lichtmodell erhalten und bunt unter weißem Neonlicht beleuchtet erscheinen. Diesen Effekt kann man tatsächlich erzielen, indem man mit `glEnable(GL_COLOR_MATERIAL);` die bereits definierte Farbe einem oder mehreren Materialparametern zuordnet. Mit `glColorMaterial(···);` kann man spezifizieren, welche Komponenten den Farbwert erhalten sollen.

5.7 Übungsaufgaben

Aufgabe 5.1 Fensterverwaltung, Farbschattierung, Tiefe und Transparenz

a) Öffnen Sie zwei Fenster als 2D-Graphik, einmal mit indizierten Farben und einmal im RGB-Modus, und ein drittes als 3D-Graphik mit Tiefenspeicher im RGBA-Modus. Zeichnen Sie mehrere Reihen verschiedenfarbiger Quadrate auf einen grauen Hintergrund. Was passiert, wenn die Ecken unterschiedliche Farbe haben?

b) Im dritten Fenster geben Sie den Quadraten unterschiedliche z-Werte, von 1.0 verschiedene α-Werte und zeichnen sie versetzt übereinander. Schalten Sie den Tiefenspeicher über Menü oder Tastatur (`z`-Taste) ein und aus.

Aufgabe 5.2 Farbsysteme

Zeichnen Sie einen transparenten RGB-Würfel in Gouraud Shading, in den Sie einen kleineren CMY-Würfel platzieren. Lassen Sie Ihre Szene mittels Maustasten routieren. In welcher Reihenfolge müssen die Kuben gezeichnet werden? Wie wirkt sich eine Größenänderung des inneren Kubus auf die Farbwirkung aus?

Aufgabe 5.3 Sonnensystem

Programmieren Sie eine 3D-Ansicht eines Sonnensystems mit einer Sonne und zwei Planeten, von denen einer einen Trabanten (Mond) hat. Die Planeten drehen sich um die Sonne, der Mond um einen Planeten.

Implementieren Sie ein Menü, um jeden einzelnen Körper wahlweise als Drahtgittermodell bzw. als vollen Körper darzustellen. Beachten Sie, dass die Sonne die einzige Lichtquelle ist! Geben Sie der Sonne zudem ein „abstrahlendes" Aussehen.

Aufgabe 5.4 Jahreszeiten

Um Jahreszeiten zu veranschaulichen, wie wir sie auf der Erde kennen, soll sich in einem reduzierten Sonnensystem die Erde 365mal um ihre eigene Achse drehen, während sie sich einmal um die Sonne dreht. Die Erdachse soll um 23,5^o von der Senkrechten zur Ebene der Erdbahn abweichen (Hinweis: Für die Neigung der Erdachse sollte man geschickt mit `glPushMatrix();` und `glPopMatrix();` umgehen, denn die Nordhalbkugel der Erde st nicht ganzjährig der Sonne zugeneigt!). Der Mond umkreist die Erde während eines Jahres zwölf mal. Zeichnen Sie Erdachse und Gradnetz auf die Erde, um die oben beschriebene (gegenüber den tatsächlichen astronomischen Bahnkurven stark vereinfachte) Rotation darzustellen.

Die zentrale Sonne stellt eine emittierende Lichtquelle dar, die auf der Erde einen hellen Lichtreflex erzeugt, wo sie im Zenit steht. Da die Erdachse um 23,5^o von der Senkrechten zur Ebene der Erdbahn

geneigt ist, lassen sich jetzt die Jahreszeiten am Zenitstand der Sonne erkennen. Der Mond wird auf seiner Kreisbahn zwölf Mal zum Vollmond.

Aufgabe 5.5 Spotlight

Schreiben Sie ein Programm, das die Wirkung von Spotlights auf grobe und feine Flächenrasterungen zeigt. Dazu bewegen Sie zwei Dreiecke flach in einem begrenzenden Quadrat hin und her und beleuchten diese Szene von oben mit einem Spotlight. Unterteilen Sie eines der beiden Dreiecke in kleinere Teildreiecke. Definieren Sie die Eigenschaften des Spotlights (siehe `glLightf(···);`, `GL_SPOT_CUTOFF, GL_LINEAR_ATTENUATION`*), so dass nur ein Teil der Szene beleuchtet wird und das Licht nach aussen linear schwächer wird.*

Wenn die Dreiecke nacheinander in den Einflussbereich des Spotlights kommen, soll das unterschiedliche Verhalten sichtbar werden. Dazu ist es hilfreich, wenn der Kegel des Spotlights so klein ist, dass er ein Dreieck nie vollständig beleuchtet.

Aufgabe 5.6 Weihnachtsbeleuchtung

Bauen Sie eine Kulisse (e.g. Theater, Musikbühne, plattes Land mit Haus und Himmel, gerne mit Weihnachtsbeleuchtung) und bewegen Sie mindestens eine Lichtquelle in Ihrer Szene auf Tastendruck (e.g. Sonne zieht ihre Bahn am Himmel, Silvesterraketen beleuchten die Nacht), so dass Ihre Kulisse aus unterschiedlichen Richtungen ausgeleuchtet wird.

Kapitel 6

Texturen

Die Entwicklung des *Texture Mapping* gehört zu den wichtigsten Beiträgen, die Edwin Catmull in den siebziger Jahren des 20. Jahrhunderts zur Computergraphik geleistet hat (neben z-Buffering und B-Splines). Die treibende Idee war, das einfache Reflexionsmodell zu verbessern, ohne Aufwand beim Lichtmodell zu betreiben. Oberflächenmaterialien sollten einfach projiziert werden.

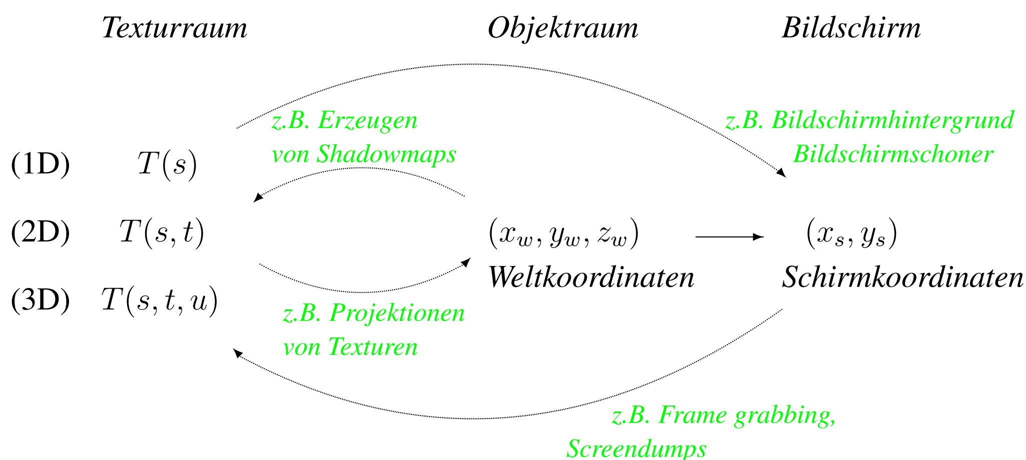


Abbildung 6.1. Texturen, Weltkoordinaten und Schirmkoordinaten.

Dreidimensionale Objekte sind in der Computergraphik (meist) aus Dreiecksflächen zusammengesetzt. Damit diese Flächen nicht wie ein kristallines Gitter erscheinen, wird mit dem Phong Shading bereits eine Manipulation an den Normalen vorgenommen. Texture Mapping ist eine weitere Methode, die Farbe auf den Oberflächen zu manipulieren, ohne die Anzahl oder Lage der Punkte zu verändern. Dadurch erscheinen computergenerierte Bilder detailreicher und realistischer, ohne dass das zugrundeliegende Modell selbst verfeinert werden muss. Dazu werden (meist) zweidimensionale

Bilder, sogenannten *Texturen*, auf diese Oberflächen abgebildet. Sie können dabei auch die Oberflächeneigenschaften verändern, die von den Algorithmen der Lichtmodelle benutzt werden.

Abbildung 6.2. Links wird die Textur wie eine Tapete aufgebracht, im rechten Bild erscheint die Textur von der Kugel reflektiert.

Ausführung der Texturabbildung	Attribute oder Parameter des Objekts
(klassisches) Texture Mapping	Diffuse Reflexion = Oberflächenfarbe
Environment Mapping	Spiegelnde Reflexion
Bump Mapping	Normalenfeld
Frame Mapping	Lokales Vektorbündel

Die Abbildungsfunktionen unterteilt man danach, welche Attribute oder Parameter eines Objekts verändert werden sollen. Einerseits ist das die Oberflächenfarbe, also der diffuse Reflexionskoeffizient eines Materials. Dies entspricht am ehesten dem Aufkleben einer Fototapete auf einem beliebig ausgedehnten, gekrümmten Objekt. Die Schwierigkeit hierbei liegt in der Art der Projektion, die eine Verzerrung des Bildes hervorrufen und zu unliebsamen Effekten führen kann, wenn das Bild zu klein, die Auflösung im Vordergrund zu gering ist oder der Winkel der Fläche zur Textur ungünstig wird.

Eine weitere elegante Methode besteht darin, die spiegelnde Reflexion zu manipulieren (*Reflection Mapping* oder *Environment Mapping*). Die Textur wird so auf das Objekt abgebildet, als ob sie auf einer größeren, das Objekt umgebenden Fläche, z.B. einem Quader, einem Zylinder oder einer Kugel angebracht wäre und von einer perfekt spiegelnden Oberfläche des Objekts reflektiert würde. Diese Fläche ist nur eine Hilfsfläche und tritt nicht beim Rendern als Objekt in Erscheinung.

Das Normalenfeld zu manipulieren, wird als *Bump Mapping* bezeichnet, und geht auf Jim Blinn (1978) zurück. Gleich ein ganzes lokales Vektorbündel wird in den Algorithmen des *Frame Mapping* manipuliert, das auf James T. Kajiya (CalTech, Pasadena, CA, 1985) zurück geht und anisotrope Oberflächen sowie deren Reflexion und Refraktion aufgrund elektromagnetischer Modellvorstellungen berücksichtigt.

6.1 Eindimensionale Texturen

Streifenmuster lassen sich durch Wiederholen einer eindimensionalen Texturmaske erzeugen, sind daher billig in Speichplatz und -zugriff. Sie können auch für kompliziertere Effekte verwendet werden, indem bei der Abbildung auf eine Fläche geeignet mit der Textur umgegangen wird. In der *Thin film technique* modelliert man die Lichtbrechung an einer das Objekt einhüllenden, dünnen transparenten Schicht, einem Film, einer Seifenhaut. Dabei ist die Intensität (nach Farben aufgeteilt)

$$I_{r(g,b)} = 0.5\left(1 - \cos\frac{2\pi R}{\lambda_{r(g,b)}}\right)$$

mit einer Verzögerungsfunktion (Retardation) $R = 2\,\eta\,d\cos\theta$ modelliert, wobei d die Schichtdicke, θ der Brechungswinkel und $\eta = \sin\phi/\sin\theta$ der Brechungsindex ist. Da die Stärke der Lichtbrechung geringfügig von der Wellenlänge λ des Lichts abhängt (blaues Licht erfährt eine stärkere Ablenkung als rotes), heißt das bei konstanter Schichtdicke und konstantem Brechungsindex η, dass die Farbe eine Funktion des Einfallswinkels ϕ ist. Das Farbspektrum des Interferenzmusters speichert man als eindimensionale Textur.

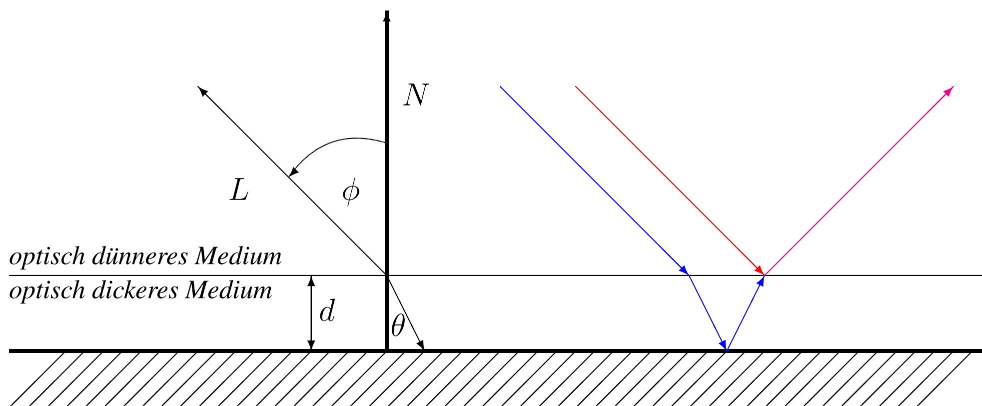


Abbildung 6.3. Lichtbrechung und Verzögerung der Ausbreitungsgeschwindigkeit.

Ist die Lichtposition L konstant, aber die Oberfläche in Betrachterrichtung unterschiedlich geneigt, so variiert das Interferenzmuster mit der Normalenrichtung N. Variiert man in der Abbildungsfunktion die Schichtdicke, erzeugt man den Eindruck eines Gels auf der Oberfläche. In Abbildung 6.3 sind links die Vektoren und Winkel und rechts schematisch die Brechung und Reflexion des blauen Anteils und die Interferenz mit dem direkt reflektierten Rotanteil des Lichts gezeigt, um die Verschiebung der Farbe durch unterschiedlich starke Verzögerung zu illustrieren. Abbildung 6.4 zeigt ein Interferenzmuster unterschiedlich stark abgelenkter roter, grüner und blauer Lichtanteile und die Auswirkung der beschriebenen Methode auf gekrümmte Flächen.

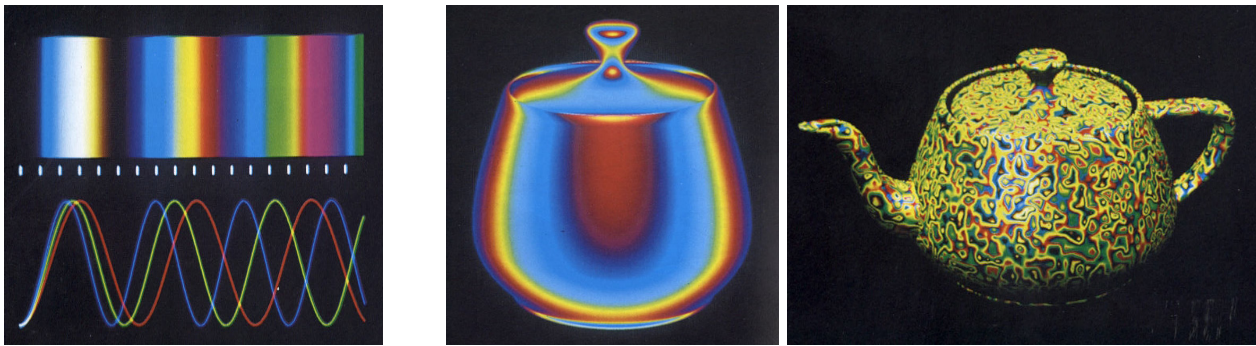


Abbildung 6.4. Links ist das Interferenzmuster dargestellt, rechts die Auswirkung auf Gefäße mit konstanter und variabler Schichtdicke d ***(aus [Wat90] Alan Watt: Fundamentals of Three dimensional Computer Graphics).***

6.2 Zweidimensionale Texturen

Wie bildet man eine 2D Textur, also ein meist rechteckiges Bild, auf eine beliebig im 3D Raum eingebettete 2D-Oberfläche ab? Die Problematik ist der des Tapezierens sehr ähnlich: auch hier kann ein Raum mit Dachschrägen, Fenstergauben oder Ofenrohren bei gemusterten Tapeten beliebig kompliziert werden. Auch Geographen und Globenbauer haben das Problem des wechselseitigen Abbildens einer annähernden Kugeloberfläche auf ebene Karten, für die es diverse Projektionsverfahren gibt. Projektionen von kugeligen auf ebene Flächen können prinzipiell nicht gleichzeitig längen- und winkeltreu sein. Ein Beispiel für eine winkeltreue und daher in der Seefahrt gern benutzte Abbildung der Erde auf eine (Welt-)Karte ist die Mercator-Projetion. Es wird eine zylindrische stereographische Projektion vorgenommen, bei der man beispielsweise am Äquator einen Zylindermantel um den Erdball legt und orthogonal zur Zylinderachse die Punkte der Erdoberfläche auf den Zylindermantel projiziert. Das hat allerdings extreme Verzerrungen in den nördlichen und südlichen Breiten zur Folge, die beispielsweise Grönland größer als Südamerika erscheinen lassen. Eine flächentreue Abbildung ist die Peters-Projektion (nach Arno Peters, 1974), ebenfalls eine Schnittzylinderprojektion, bei der die Schnittkreise jewels den Breitenkreisen angepasst werden.

6.2.1 UV-Mapping

In der Mathematik hat man den Begriff *Atlas* für eine Menge von Karten auf einer Mannigfaltigkeit übernommen, wobei eine Karte eine Bijektion (in beide Richtungen eindeutige Abbildung) von einer Untermenge der Mannigfaltigkeit meist auf einen reellen, endlichdimensionalen Vektorraum $\mathbb{R}^n$ ist. In der Computergraphik spricht man auch von einem Atlas, dem Texturatlas. Damit sind viele kleine Texturen und die Abbildungsvorschriften im Sinne des sogenannten UV-Mapping gemeint, bei der die 2D Textur mit den Koordinaten u und v parametrisiert wird (aber auch s und t sind gebräuchlich). In der Regel gibt es für eine 3D-Szene mehrere Bilder, die als Texturen verwendet werden. Die 3D-Szene besteht dabei aus Polygonen. Man kann nun jedem Polygon eine Textur zuweisen. Natürlich können mehrere Polygone die selbe Textur verwenden. Jedem Vertex (x_w, y_w, z_w) eines Polygons wird eine u und eine v-Koordinate zugewiesen, die angeben, welchem Texel (Bildpunkt in der Textur) dieser Eckpunkt des Polygons zugeordnet wird. Um die Position innerhalb einer Textur zu bestimmen, hat sich ein Koordinatensystem eingebürgert, bei dem (0,0) die linke untere Ecke und (1,1) die rechte obere Ecke der Textur bezeichnet; die beiden anderen Ecken sind naturgemäß (1,0) und (0,1). Die Texturkoordinaten sind in der Regel Fließkommazahlen. Wenn eine Texturkoordinate 0 unterschreitet oder 1 überschreitet, also den Rand der Textur hinausweist, so wird die Textur meist periodisch wiederholt.

Die einfachste Abbildung zwischen der Textur und den Schirmkoordinaten des Vertex (x_s, y_s) ist die lineare Interpolation. Die Texturkoordinaten werden entlang der schon vom 3D- in den 2D-Raum transformierten Randlinien des Polygons linear interpoliert. Dann werden sie entlang einer Bildschirmzeile von Randlinie zu Randlinie linear interpoliert. An jedem Pixel wird jetzt der Farbwert des zu den interpolierten (u, v)-Koordinaten gehörenden Texels übernommen. Dabei müssen gerade Linien (üblicherweise horizontal) im Bildspeicher gerendert und auf der Textur entsprechend schräge Linien abgetastet werden.

Dabei können unterschiedliche Texturkoordinaten für ein und denselben Punkt verwendet werden, wenn dieser Punkt zu verschiedenen Polygonen gehört. Oft wird aber für ein 3D-Modell aus vielen Polygonen eine einzige Textur für das ganze Modell verwendet, sodass jeder Punkt des Modells eindeutige Texturkoordinaten hat, weil dieses Format für hardwarebeschleunigte 3D-Grafik besonders günstig ist. In 3D-Modellierungssoftware wird zum Bearbeiten der Objekte ein UV-Mapping angeboten, bei dem der Nutzer die Textur auf dem Objekt geeignet verzerren kann.

6.2.2 Perspektivkorrektur

Bei orthographischer Projektion genügt das oben beschriebene Verfahren. Aber bei perspektivischer Darstellung von Polygonen, die sich merklich in Sichtrichtung verkürzen, führt das Verfahren zu visuell unbefriedigenden Resultaten, weil die Texturkoordinaten nach der Projektion interpoliert werden. Damit wird nicht berücksichtigt, dass eine Strecke im weiter entfernten Teil des projizierten Polygons einer größeren Strecke im originalen Polygon im 3D-Raum entspricht. Die Zuordnung von Texturkoordinaten zu Punkten im dreidimensionalen Raum muss sich ändern, wenn sich die Betrachterposition

ändert. Um dieses Problem zu lösen, werden meist anstatt der Texturkoordinaten u und v die Werte von u/z und v/z und auch $1/z$ linear interpoliert, wobei z die Koordinate im 3D-Raum in Sichtrichtung ist (z bzw. $1/z$ muss daher zu jedem projizierten Punkt des Polygons gespeichert werden). Um für einen Pixel die Texturkoordinaten zu berechnen, müssen nun Divisionen ausgeführt werden:

$$u' = (u/z)/(1/z)$$
$$v' = (v/z)/(1/z)$$

Weil Divisionen relativ langsame Operationen sind, werden sie meist nicht bei jedem Pixel gemacht; stattdessen werden u und v nur bei wenigen Pixeln, die auf das Polygon gleichmäßig verteilt sind, so berechnet. Bei allen anderen Pixeln werden die Werte von u und v zwischen denen jener Pixel interpoliert. So kann man die störenden Effekte stark reduzieren, ohne dass all zu viel Rechenleistung dafür aufgewendet werden muss.

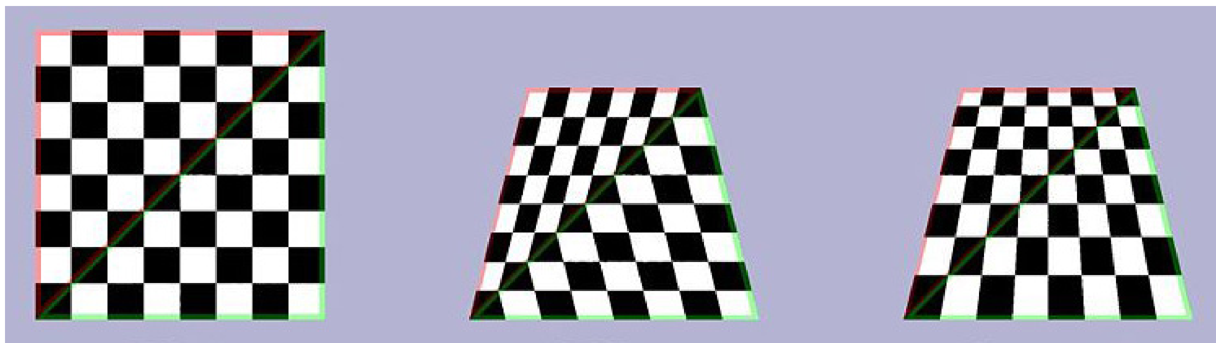


Abbildung 6.5. Eine Schachbretttextur wird auf ein Quadrat abbgebildet, das perspektivisch geneigt wird. Die Projektion (mitte) zeigt deutlich den Fehler, der durch einfache affine Transformation der Textur auf die Schirmkoordinaten entsteht. Rechts wurde der Fehler korrigiert.

6.2.3 Entfaltung des Polygonnetzes

Bei einer allgemeinen Parametrisierung, die auf Paul S. Heckbert (1986) zurück geht, werden alle Flächen in Dreiecke zerlegt. Fur jeden Punkt auf dem Objekt gibt es eine lineare Beziehung

$$(x_w, y_w, z_w) = (A_x u + B_x v + C_x, A_y u + B_y v + C_y, A_z u + B_z v + C_z)$$

mit den Texturkoordinaten u und v. Für Polygone mit mehr als drei Ecken werden nichtlineare Transformationen nötig. Dieser direkte mathematische Zugang kann zudem auf topologische Probleme bei komplizierteren Objekten führen.

Um sich über die Topologie Klarheit zu verschaffen, kann man beispielsweise das Polygonnetz, das sogenannte *Mesh*, entfalten. Nachbarschaftsbeziehungen müssen dabei gesondert gespeichert werden.

Für ebene Netze ist das Texturieren jetzt sehr einfach, aber an Kanten, die erst nach dem Mapping miteinander identifiziert werde, treten Unstetigkeiten in der Textur auf (siehe die farbigen Kanten in Abbildung 6.6).

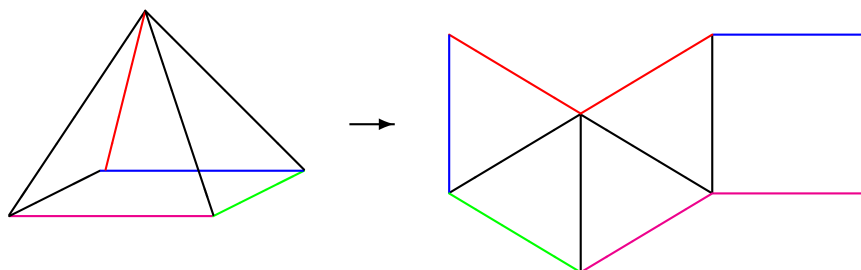


Abbildung 6.6. Ein 3D Objekt wird in ein ebenes Gitter auseinander gefaltet. Gleichfarbige Linien werden dabei miteinander identifiziert.

6.2.4 Zweiteiliges Abbilden

Eine andere Methode ist das zweiteilige Abbilden, bei dem die Textur vor dem Abbilden dem Objekt grob angepasst wird, um es in einem zweiten Schritt (*shrink to fit*) auf das Objekt zu projizieren. Man bezeichnet diese Techniken als S-Mapping oder O-Mapping, S steht dabei für *Surface*, O für *Objekt*.

1. S-Mapping: $T(u,v) \rightarrow T'(x_i,y_i,z_i)$ Abbilden auf eine einfache 3D-Zwischenablagefläche
2. O-Mapping: $T'(x_i,y_i,z_i) \rightarrow O(x,y,z)$ Abbilden auf das Objekt

Dabei kann die Zwischenablage beispielsweise ein Zylinder sein, auf den die Textur einer Schlangenhaut abgebildet wird. Im zweiten Schritt wird die zylindrische Textur der unterschiedlichen Dicke und (leichten) Biegungen der Schlange angepasst. Dabei werden die u und v Koordianten auf Zylinderkoordinaten

$$\begin{aligned} x_i &= \rho \cos \phi \\ y_i &= \rho \cos \phi \\ z_i &= d\, z \end{aligned}$$

als Interim-Koordinaten geeignet abgebildet. Zur Skalierung werden der Radius ρ und die Länge d verwendet, ansonsten wird die Textur $T(u,\cdot)$ und $T(\cdot,v)$ geeignet auf den Winkel ϕ und die z Koordinate abgebildet und erzeugt so eine Textur T', die über x_i, y_i und z_i parametrisiert ist. Gängige andere Zwischenablagen sind gedrehte, beliebig orientierte Ebenen, Kugeln oder Würfel.

Für das O-Mapping, also das finale Abbilden auf das Objekt, hat man ebenfalls verschiedene Möglichkeiten: Die Projektionsstrahlen können senkrecht zur Zwischenablage im Punkt (x_i, y_i, z_i) oder senkrecht zum Objekt in (x, y, z) oder entlang eines Strahls, auf dem der Punkt (x, y, z) und das Zentrum des Objekts liegen, gewählt werden. Wichtig ist, das die Abbildung eineindeutig, also bijektiv ist. Der erste Schnittpunkt des Projektionsstrahls mit dem Objekt ordnet die Farbe des Texels den Objektkoordinaten zu.

Beim *Environment Mapping* wird der Projektionsstrahl vom Betrachterstandpunkt abhängig gemacht, indem man einen an der Oberfläche reflektierten Strahl berechnet. Bezogen auf die Oberflächennormale in den verschiedenen Objektpunkten ergeben sich für einen festen Betrachter variiernde Einfallswinkel und entsprechende Ausfallswinkel. Die Schnittpunkte der reflektierten Strahlen mit der Zwischenablage ordnen den Objektpunkten (x, y, z) die Texturfarben $T'(x_i, y_i, z_i)$ zu (siehe Abbildung 6.2).

6.3 Dreidimensionale Texturen

Dreidimensionale Texturen stellen ganze Blöcke aus äquidistanten Farbinformationen in alle drei Raumrichtungen dar und können beispielsweise aus Schichten von Einzelbildern zusammengeetzt sein. Solche Texturen findet man häufig in der Medizintechnik, wo beispielsweise in der Computertomographie (CT) oder bei Magnetresonanzbildern (MRI) solche Volumen aus einzelnen gemessenen Schnittbildern zusammengesetzt werden.

$$(x_w, y_w, z_w) \mapsto T(x_w, y_w, z_w) \equiv \text{id}$$

Das Objekt (in Weltkoordinaten) wird aus dem Texturblock herausgeschnitten. Am Beispiel einer CT-Aufnahme eines Schädels sind die Objekte Haut, Knochen oder Hirn. Durch die Identitätsabbildung erscheinen diese Objekte mit der für sie passenden Oberflächenstruktur. So kann man sie aus dem gesamten Block der (undurchsichtigen) CT-Volumendaten herauslösen und einzeln darstellen.

Dieses Verfahren ist extrem speicherplatzintensiv, hat aber gleichzeitig einen geringen Abbildungsaufwand. Daher lohnt es für Animationen auf Highend Graphikmaschinen (mit großem Arbeitsspeicher), bei denen man Objekte mit statischen Texturfeldern verschneidet.

Will man den Speicherplatz für eine 3D Textur geringer halten als bei der Identitätsabbildung nötig (hier müsste die Textur die Objektgröße umfassen), muss die Textur entsprechend skaliert werden. Werden Berechnungen zum Erzeugen eines Texels nötig, spricht man von *prozeduralem* Texturieren. Ein einfaches Beispiel besteht in einer Textur für eine Holzmaserung: leicht exentrisch ineinander geschachtelte Zylinder in abwechselnd hellen und dunklen Brauntönen können eine beliebig große Textur erzeugen. Wenn jetzt noch die Mittelachse leicht gegen eine Hauptachse des Objekts geneigt ist, entsteht auf den Objektflächen je nach Raumrichtung ein Muster aus konzentrischen Kreisen oder parabelähnlichen Streifen.

Abbildung 6.7. Wenn ein Objekt aus Massivholz nicht wie furniert wirken soll, muss die Holzmaserung dreidimensional vorliegen. Die Oberflächenfarbe entsteht aus dem Schnitt der Fläche mit einer dreidimensionalen Textur.

6.4 Bump Mapping

Unter *Bump Mapping* versteht man das Verändern der Oberflächenfarben auf einem glatten Objekt zur Erzeugung von Oberflächendetails. Die am weitesten verbreitete Methode ist die Veränderung der Oberflächennormale.

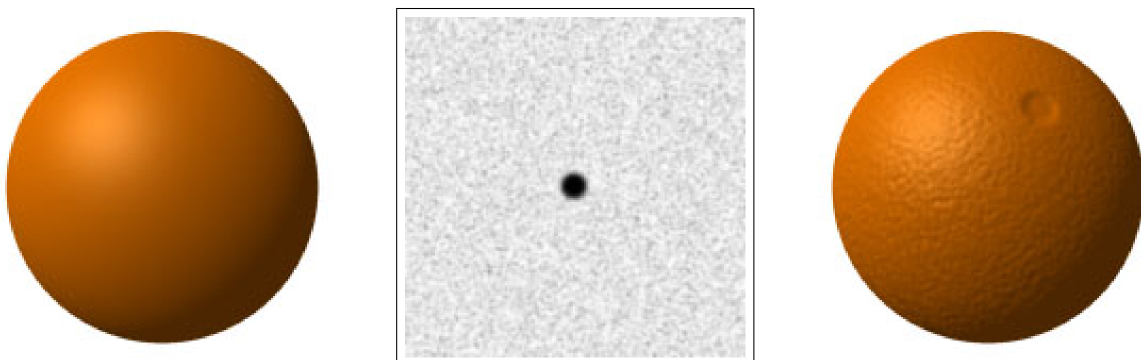


Abbildung 6.8. Auf einer orangen Kugel wird mit der Textur (mitte) das Vektorfeld der Normalen verändert, so dass der Eindruck einer detaillierten Oberfläche einer Orange entsteht.

Dabei wird $N' = N + D$ als gestörtes Normalenfeld aus dem tatsächlichen Normalenfeld N und einem *Displacement* D berechnet. Im Folgenden wird gezeigt, wie man dieses D aus einer Textur berechnet.

Sei $N = \frac{\partial O}{\partial s} \times \frac{\partial O}{\partial t}$ das äußere Richtungsfeld, also ein (nicht normiertes) Vektorfeld von Oberflächennormalen, die sich aus dem Kreuzprodukt der Tangenten an den Oberflächenpunkt $O(s,t)$ berechnen. Der Einfachheit halber ist die Oberfläche mit Koordinaten s und t in gleicher Weise wie die Textur parametrisiert. Jetzt ist O' eine Oberfläche, deren Normalenfeld mit einer Textur $T(s,t)$ gestört wurde.

$$O'(s,t) = O(s,t) + T(s,t)\frac{N}{|N|}$$

Das gestörte Normalenfeld $N'(s,t) = \frac{\partial O'}{\partial s} \times \frac{\partial O'}{\partial t}$ berechnet sich also als Kreuzprodukt aus den gestörten Oberflächentangenten, wobei O' zunächst nach der Produktregel differenziert wird, was $\frac{\partial O'}{\partial s} = \frac{\partial O}{\partial s} + \frac{\partial T}{\partial s}\frac{N}{|N|} + T\frac{\partial}{\partial s}\frac{N}{|N|}$ und $\frac{\partial O'}{\partial t} = \frac{\partial O}{\partial t} + \frac{\partial T}{\partial t}\frac{N}{|N|} + T\frac{\partial}{\partial t}\frac{N}{|N|}$ ergibt. Daraus berechnet sich

$$\Rightarrow N'(s,t) = \frac{\partial O}{\partial s} \times \frac{\partial O}{\partial t} \quad + \quad \frac{\partial T}{\partial s}\left(\frac{N}{|N|} \times \frac{\partial O}{\partial t}\right) + \frac{\partial T}{\partial t}\left(\frac{N}{|N|} \times \frac{\partial O}{\partial s}\right) \quad + \quad \frac{\partial^2 T}{\partial s \partial t}\left(\frac{N}{|N|} \times \frac{N}{|N|}\right).$$

Der vordere Term ist das ursprüngliche Normalenfeld. Der hintere Term dagegen entfällt, da das Kreuzprodukt eines Vektors mit sich selber Null ergibt.

$$\Leftrightarrow N' = N + D = N \quad + \quad \frac{\partial T}{\partial s}\left(\frac{N}{|N|} \times \frac{\partial O}{\partial t}\right) + \frac{\partial T}{\partial t}\left(\frac{N}{|N|} \times \frac{\partial O}{\partial s}\right)$$

Das gestörte Normalenfeld N' muss korrekterweise anschließend noch normiert werden. In Abbildung 6.9 wird das Verfahren für einen eindimensionalen Schnitt schematisch dargestellt.

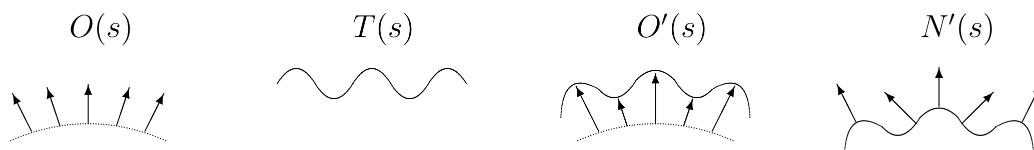


Abbildung 6.9. Eindimensionale Darstellung des Bump Mapping mit $O(s)$ und zugehörigem Normalenfeld; einer Textur $T(s)$, der Bump Map; einer gestörten Oberfläche $O'(s)$ und einem neuen Normalenfeld $N'(s)$.

6.4.1 Displacement Mapping

Im Gegensatz zum Bump Mapping verändert das *Displacement Mapping* die geometrischen Punkte. Es wird als nicht nur über die manipulierten Normalen die lokale Farbe verändert, sondern das Verschieben einzelner Punkte erzeugt das Verändern der Normalen. Am Rand eines solchen veränderten Objekts ist der Unterschied deutlich an der Silhouette erkennbar.

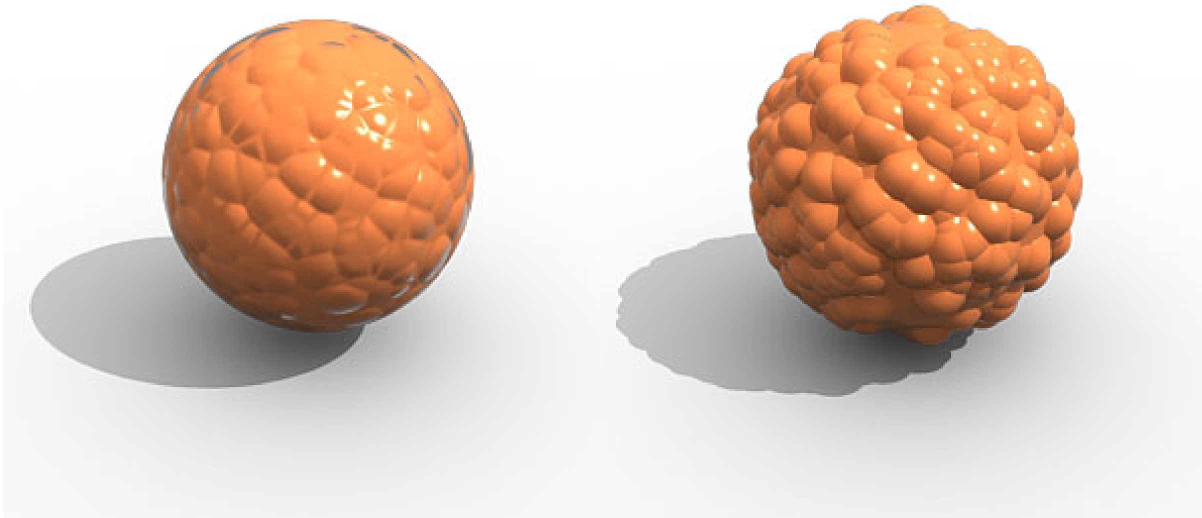


Abbildung 6.10. Auf einer orangen Kugel wird links das Vektorfeld der Normalen verändert, rechts werden die Punkte selbst manipuliert.

6.5 Interpolationen und Skalierungen

Die bisher beschriebenen Verfahren nehmen vereinfachend an, dass jeder Pixel exakt einem Texel zugeordnet werden kann. Betrachtet man aber sowohl Pixel als auch Texel als Punkte ohne Ausdehnung, so ist dies im Allgemeinen nicht der Fall. Vielmehr liegen die Texturkoordinaten eines Pixels in der Regel zwischen mehreren Texeln. Es gilt also, zu entscheiden, wie aus den Farbwerten der umliegenden Texel der Farbwert für den Pixel gewonnen wird: Man benötigt ein geeignetes Interpolationsverfahren.

Das einfachste und schnellste solche Verfahren besteht darin, einfach den nächstliegenden Texel auszuwählen; man nennt dies *nearest neighbor* oder auch *point sampling*, da es sich um punktweise Stichproben handelt. Beim aufwändigeren bilinearen Filtern wird der gesuchte Farbwert aus den vier umliegenden Texeln in Abhängigkeit ihrer Entfernung interpoliert, man nennt dies einen *Tent-Filter*. Noch aufwändigere Filter, etwa der Gauß-Filter, ziehen weitere Texel in die Berechnung mit ein oder gewichten die Entfernung anders. Da ungeeignete Interpolationsverfahren zu unerwünschten Alias-Effekten führen wie beispielsweise Moiré-Effekte, muss ein Kompromiss zwischen Geschwindigkeit und Artefaktbildung gefunden werden.

Das übliche, oben beschriebene Texture Mapping wird angewandt, solange der Rasterabstand der Pixel kleiner als jener der Texel ist: einem beliebigen Pixel wird höchstens ein Texel zugeordnet, mehreren benachbarten Pixeln kann durchaus der gleiche Texel zugeordnet werden. Ist der Rasterabstand der Pixel jedoch größer als jener der Texel, so entspricht einem Pixel gleich ein ganzer Bereich der Textur. Der Farbwert des Pixels müsste jetzt als Mittelwert sämtlicher Texel gebildet werden, was

sehr aufwändig und daher nicht praktikabel ist: für einen einzigen Pixel müssten viele Rechenoperationen ausgeführt werden. Außerdem führt diese Mittelung auf seltsame, unbeabsichtigte Muster, die durch diskontinuierlich sich ändernde Farbwerte benachbarter Pixel entstehen (*aliasing*).

6.5.1 Mipmapping

Stattdessen verwendet man *Mipmaps* (auch MIP maps genannt). Diese enthalten neben der Bitmap der Originaltextur Kopien der Textur mit abnehmender Größe, sogenannte Detailstufen oder *level of detail*, LOD. Die Buchstaben MIP sind ein Acronym für die lateinische Phrase *multum in parvo*, was übersetzt soviel bedeutet wie *viel auf kleinem Raum*. Das zusätzliche Abspeichern der Detailstufen braucht zwar mehr Speicherplatz, zahlt sich aber durch bessere Qualität der Bilder ***und*** schnellere Berechnung aus.

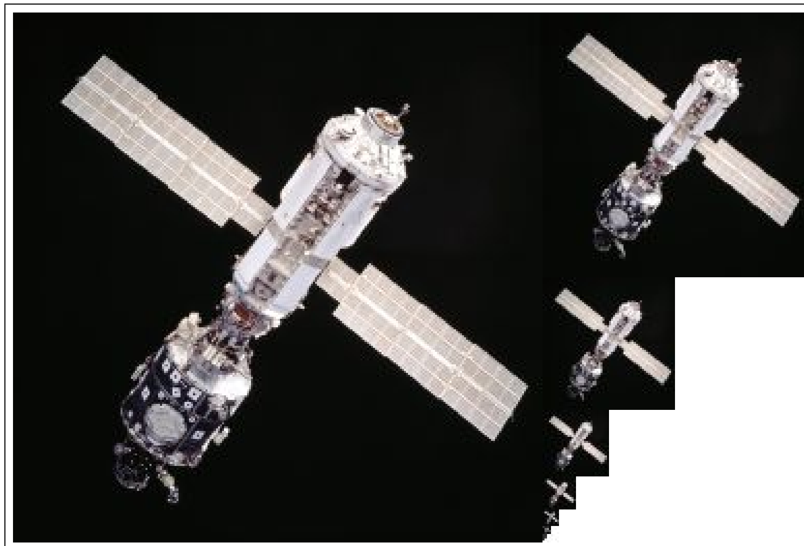


Abbildung 6.11. Eine originale Textur und ihre Mipmapreihe mit halbierter Kantenlänge.

Das *Mipmapping* wurde Anfang der achtziger Jahre von Lance Williams erfunden und in seiner Veröffentlichung *Pyramidal parametrics* von 1983 beschrieben. Es stellt eine Methode dar, wie Texturen vorgefiltert und optimal kleiner skaliert werden, um *Aliasing* Effekte beim Texture Mapping zu minimieren und auch den Aufwand beim Texturieren gering zu halten, wenn diese Objekte durch größeren Abstand zum Betrachter kleiner erscheinen.

Texturen sind typischerweise quadratisch und haben eine Zweierpotenz als Kantenlänge. Jedes Bitmap Bild einer Mipmap Reihe ist eine eigene Version der ursprünglichen Textur, aber in der Größe reduziert. Hat die Basistextur eine Größe von 256 auf 256 Pixel, so hat die assoziierte Mipmap eine Reihe von acht Bildern, deren Kantenlänge gegenüber dem vorigen Bild jeweils halbiert wurden, also 128×128 Pixel, 64×64, 32×32, 16×16, 8×8, 4×4, 2×2, 1×1, also ein einziges Pixel. Wenn in der Szene eine Fläche von 40×40 Pixeln mit der Textur belegt werden soll, weil das Objekt in großer

Abbildung 6.12. Ablage einer Textur im Texturspeicher, nach Farbkanälen separiert. In natürlicher Weise ergibt sich die halbierte Kantenlänge für die nächstkleinere Textur.

Entfernung oder überhaupt nur so klein erscheint, wählt man aus dieser Reihe die größte Detailstufe aus, die noch gerade den Zustand herstellt, bei der der Pixel kleiner als der Texel ist, also die Bitmap 32×32, und arbeitet darauf wie auf der Originaltextur. Man kann aber auch zwischen den 64×64 und 32×32 Mipmaps interpolieren und dieses vorgefilterte Resultat benutzen. Als Maß für die Auswahl einer Texturgröße dient ein Parameter

$$D = \max \left\{ \left(\left(\frac{\partial s}{\partial x} \right)^2 + \left(\frac{\partial t}{\partial x} \right)^2 \right)^{\frac{1}{2}}, \left(\left(\frac{\partial s}{\partial y} \right)^2 + \left(\frac{\partial t}{\partial y} \right)^2 \right)^{\frac{1}{2}} \right\}$$

In Kombination mit der bilinearen Filterung erhält man so eine trilineare Filterung. Die benötigte Zeit zum Rendern wird reduziert, denn die Anzahl der Texel, die berücksichtigt werden müssen, reduziert sich drastisch gegenüber der ursprünglichen Textur. Auch das Skalieren nach oben und unten geschieht mit Mipmap effizienter. Nebenbei werden auch noch Artefakte deutlich reduziert. Demgegenüber steht ein um ein Drittel größerer Speicherbedarf bezogen auf die Originaltextur, da die Bildfläche quadratisch abnimmt und die Reihe

$$\frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \cdots = \sum_{n=1}^{\infty} \left(\frac{1}{2^n} \right)^2 = \frac{1}{3}$$

gegen ein Drittel konvergiert. Algebraisch verifiziert man dieses Ergebnis, weil man weiß, dass

$$\sum_{n=1}^{\infty}\left(\frac{1}{2^n}\right) = \lim_{n\to\infty}\frac{2^n-1}{2^n} = 1.$$

Diese Summe setzt sich zusammen aus den Teilsummen der geraden und ungeraden Potenzen von $\frac{1}{2}$, wobei die Summe der ungeraden Potenzen um 2 erweitert doppelt so groß sein muss.

$$1 = \sum_{n=1}^{\infty}\left(\frac{1}{2^n}\right) = \sum_{n=1}^{\infty}\frac{1}{2^{2n}} + \sum_{n=1}^{\infty}\frac{1}{2^{2n-1}} = \sum_{n=1}^{\infty}\frac{1}{2^{2n}} + \sum_{n=1}^{\infty}\frac{2}{2^{2n}} = \frac{1}{3} + \frac{2}{3}.$$

Der Einsatz von Mipmaps in Verbindung mit *point sampling* reduziert Alias-Effekte bereits stark. In Verbindung mit aufwändigeren Filtern und Interpolation zwischen den Detailstufen können sie auf ein Minimum reduziert werden.

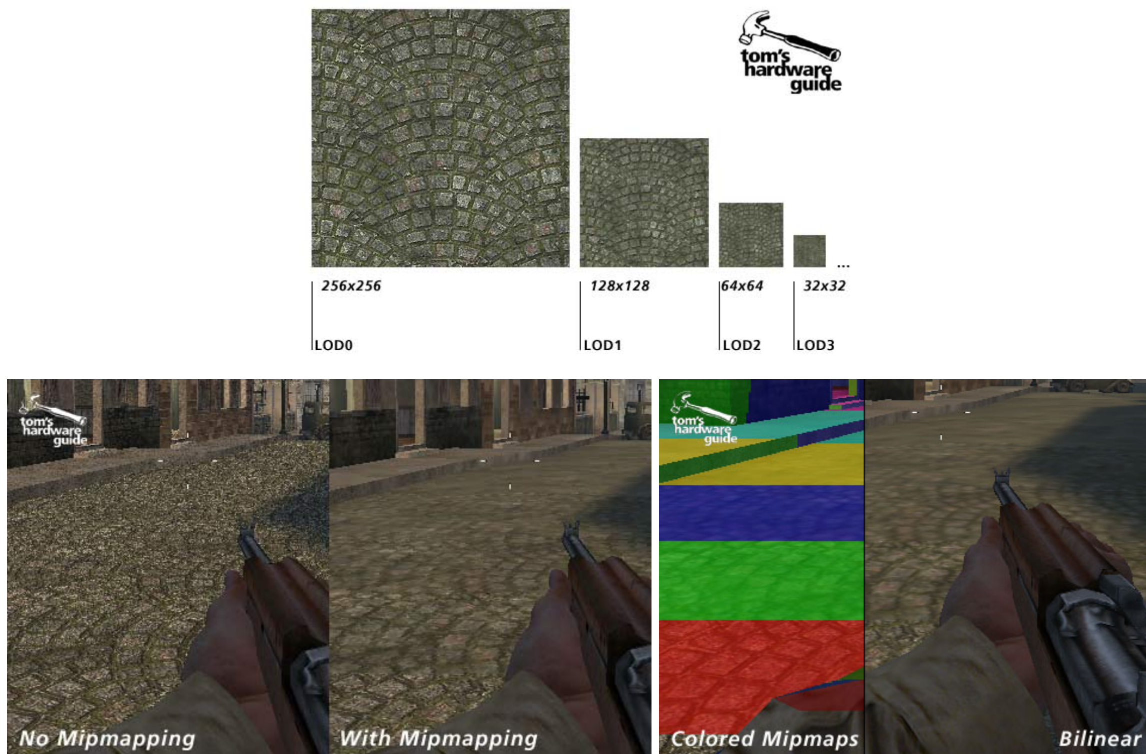


Abbildung 6.13. Die vorgefilterten Texturen des Straßenpflasters werden der Größe des Objekts entsprechend ausgewählt (siehe die farbkodierten Streifen). Durch bilineares Interpolieren werden Moiré-Effekte und Aliasing vermieden.

6.5.2 Ripmapping

In vielen Fällen ist es günstiger, anisotrop zu filtern und die Texturen zunächst nur in einer Richtung, danach in der anderen Richtung zu halbieren. Diese sogenannten Ripmaps brauchen allerdings viermal soviel Speicherplatz, wie die Originaltextur. Zum Beispiel wird eine 8×8 Pixel große Textur mit

einer Ripmap Reihe von 8×4, 8×2, 8×1, 4×8, 4×4, 4×2, 4×1, 2×8, 2×4, 2×2, 2×1, 1×8, 1×4, 1×2, 1×1 abgespeichert. Außerdem geht die Kohärenz im Cache verloren, was den Speicherzugriff verlangsamt. Dagegen sind die Vorteile gesteigerter Bildqualität abzuwägen. Jedenfalls sind Ripmaps inzwischen unpopulär und von einem Kompromis aus anisotropem Filtern und Mipmapping ersetzt worden, bei dem höhere Auflösungen der Mipmaps richtungsabhängiges Vorfiltern bei kohärentem Cache erlauben.

6.6 OpenGL Implementierung

Das Texturieren geschieht in OpenGL in drei Hauptschritten: zunächst definiert man die Textur, dann kann man diese Textur unterschiedlich interpretieren lassen und schließlich kontrolliert man über Parameter die Art und Weise der Abbildung. Defaultmäßig ist das Texturieren nicht aktiv. Daher muss es und sogar für jede Texturrichtung einzeln mit `glEnable(GL_TEXTURE_GEN_S);` oder `glEnable(GL_TEXTURE_GEN_T);` aktiviert werden.

Definieren der Textur

Man kann ein-, zwei- oder dreidimensionale Texturen in den Texturspeicher laden. Eine zweidimensionale Textur wird beispielsweise mit `glTexImage2D(target, level, internalformat, width, height, border, format type, *pixels);` definiert (1D und 3D mit entsprechend anderen Aufrufen).

Interpretieren von Texturfragmenten

Den eigentlichen Abbildungsvorgang kann man gezielt mit `glTexEnv..(coord, pname, *params);` steuern. Hier wird festgelegt, auf welchen Farbwert sich die Textur auswirkt oder ob sie beispielsweise einem Farb- und Lichtmodell überblendet wird.

Mit `glTexParameter..(coord, pname, *params);` steuert man Parameter für das Mipmapping oder für das Verhalten beim Überschreiten der Texturgrenzen (Wiederholung der gesamtem Textur oder nur des Randes) bei.

Kontrollieren der Abbildungseigenschaften

Ob linear entlang des Sichtstrahls vom Auge oder von den Objektkoordinaten her projiziert wird, kann man mit `glTexGen..(coord, pname, *params);` festlegen.

Grundsätzlich sind alle Parameter sinnvoll voreingestellt. Daher wirkt sich bei eingeschaltetem Texturieren die zuvor geladene Textur auf alle nachfolgenden Objekte aus. Will man gezielt Texturkoordinaten auf Objektkoordinaten abbilden, wird vor dem Aufruf des Vertex beispielsweise `glTexCoord2..(s, t);` aufgerufen. Damit kann man statt einer Identitätsabbildung die Textur beliebig auf dem Objekt verzerren.

6.7 Übungsaufgaben

Aufgabe 6.1 Texturierter Würfel

Texturieren Sie einen Würfel mit sechs unterschiedlichen Bildern auf seinen Außenflächen. Wie bei einem Spielwürfel soll jeder Fläche eine Ziffer von eins bis sechs zugeordent sein. Diese Ziffer erkennt man an der Häufigkeit, mit der die Textur auf der Fläche wiederholt wird.

Aufgabe 6.2 Texturiertes Sonnensystem

Erweitern Sie Ihr Sonnensystem aus dem vorigen Kapitel um Texturen, indem Sie Ihre Erde mit einer realistischen Erdoberfläche texturieren, die Sie sich als Bild im Internet besorgen können. Erzeugen Sie die Oberfläche eines Zwergplaneten über Bump Mapping, indem Sie die Normalen einer selbst geschriebenen Routine für eine Kugel zufällig stören. Erzeugen Sie in einem Bildbearbeitungsprogramm eine verrauschte Textur und lassen Sie diese auf das Normalenfeld einer Kugel wirken.

Aufgabe 6.3 Bump Mapping

Erstellen Sie eine eigene Routine `mySphere(int nb, int nl, int bumb)`*, mit der Sie eine in* `nb` *Breitengrade und* `nl` *Längengrade unterteilte Kugel zeichnen. Der 0/1-Schalter* `bump` *gibt an, ob die Kugel mit tatsächlichen Normalen oder gestörten Normalen auf jedem Vertex versehen werden soll. Um die Normalen zu stören, generieren Sie für jede Vertexnormale drei zufällige Störfaktoren* `fx, fy, fz` $\in [0.5; 2.0]$*, mit denen Sie die Komponenten der Normale multiplizieren. Renormalisieren Sie die entstandenen Vektoren danach wieder. Erstellen Sie eine Szene, in der Sie eine glatte und eine mit Bumpmapping versehene Kugel platzieren und lassen Sie beide unter einer gemeinsamen Lichtquelle um je eine Körperachse rotieren. Implementieren Sie Tasten* `b/B` *und* `l/L`*, um die Unterteilungsfeinheiten* `nb` *und* `nl` *in sinnvollen Grenzen zu verändern.*

Aufgabe 6.4 Pflanzen

Erzeugen Sie realistisch wirkende Planzen, indem Sie sich digitale Bilder von Bäumen oder Büschen verschaffen. Lassen Sie nur die Blätter und Zweige opaque erscheinen, während die Umgebung transparent dargestellt wird. Verwenden Sie diese Texturen so auf einfachen geometrischen Objekten (beispielsweise auf zwei orthogonal sich kreuzenden Rechtecken), dass sie in ihrer Szene realistisch erscheinen.

Kapitel 7

Raytracing

Die bisher betrachteten einfachen Lichtmodelle arbeiten gut, wenn wenige, weit auseinanderliegende Objekte betrachtet werden. Sie arbeiten mit konstantem Umgebungslicht und ohne Bezug auf andere Objekte, denn es gehen nur die Position der Lichtquelle, die Normale der Objektfläche und die Betrachterposition ein. Damit ist es nicht möglich, Schlagschatten darzustellen, also das Verdecken einer Lichtquelle durch ein Objekt. Genausowenig können Spiegelungen an anderen Objekten berücksichtigt werden. Komplexere Objekte müssen mit weiteren Tricks arbeiten, um realistischer zu wirken. Beispielsweise kann man Schlagschatten bei fester Lichtquelle einmal berechnen und als Textur auf die Objekte abbilden. Dieses sogenannte *Shadowcasting* ist in der OpenGL-Version1.3 implementiert.

Das *Raytracing* ist ein möglichst vollständiges und eindrucksvolles Illuminations- und Reflexionsmodell. Es geht als *Raycasting* auf Arbeiten von Arthur Appel 1968 zurück, die für *Hidden Surface Removal* entwickelt wurden. Dazu wurde der Strahl nur bis zum ersten Auftreffen auf ein Objekt verfolgt. Das Verfahren wurde als zu aufwendig verworfen und durch den z-Buffer Algorithmus ersetzt. Erst gut zehn Jahre später war man in der Lage, mit ähnlichen Algorithmen Illumination zu berechnen (1979 Douglas Scott Kay, 1980 Turner Whitted). Dieses Verfahren wird auch als *Rekursives Raytracing* bezeichnet. Dabei wird aus

$$\left.\begin{array}{c}\text{Reflexion}\\ \text{und}\\ \text{Transmission}\end{array}\right\}\text{globale Illumination}$$

zusammengesetzt. Der wesentliche Unterschied zu den bisherigen Lichtmodellen besteht darin, dass das *Raytracing* im Objektraum arbeitet. Das bedeutet, dass immer alle Objekte im Raum und ihre Lage zueinander und zum Betrachter berücksichtigt werden müssen. Die eleganten und schnellen *z-Buffer Algorithmen* kommen nicht zur Anwendung. Erklärlicherweise werden die Rendering Algorithmen im Objektraum bedeutend langsamer, je mehr Objekte berücksichtigt werden müssen. Geschicktes Sortieren wird damit notwendig.

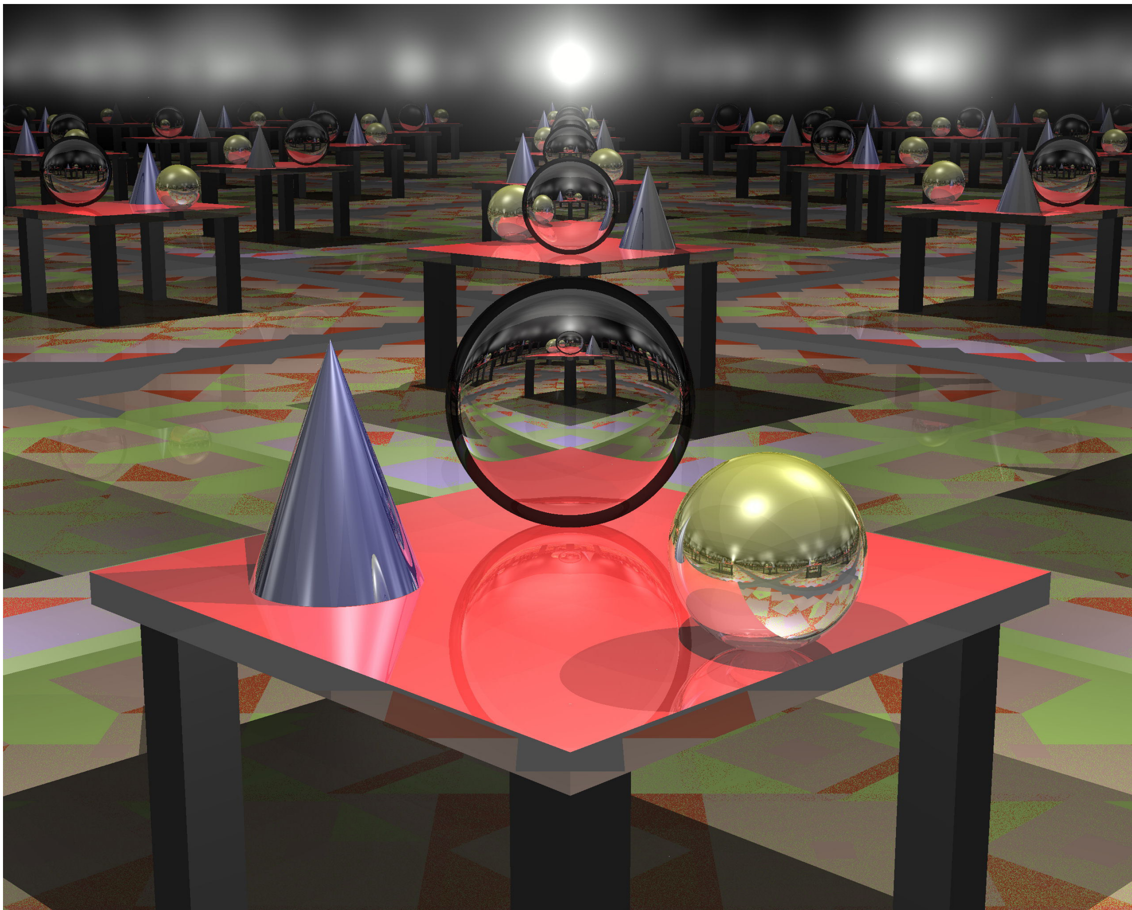


Abbildung 7.1. An Kugeln sehen Reflexion und Transmission immer besonders eindrucksvoll aus. Die Szene zeigt einen Tisch mit silbernem Konus, Glaskugel und goldener Kugel, der in einer spiegelnden Raumecke steht. Das Logo der Arbeitsgruppe *Numerical Geometry Group* am IWR ist die goldene Kugel aus diesem Bild, das mit einem Anfang der 90er Jahre am IWR entwickelten Raytracer berechnet wurde *(Autoren: Christoph Kindl, Markus Ridinger, Norbert Quien).*

Vorteile:

Von einem einzigen Modell werden

- Hidden Surface (automatischer Bestandteil)
- Transparenz (automatischer Bestandteil)
- Direkte Beleuchtung
- Indirekte Beleuchtung
- Schattenwurf

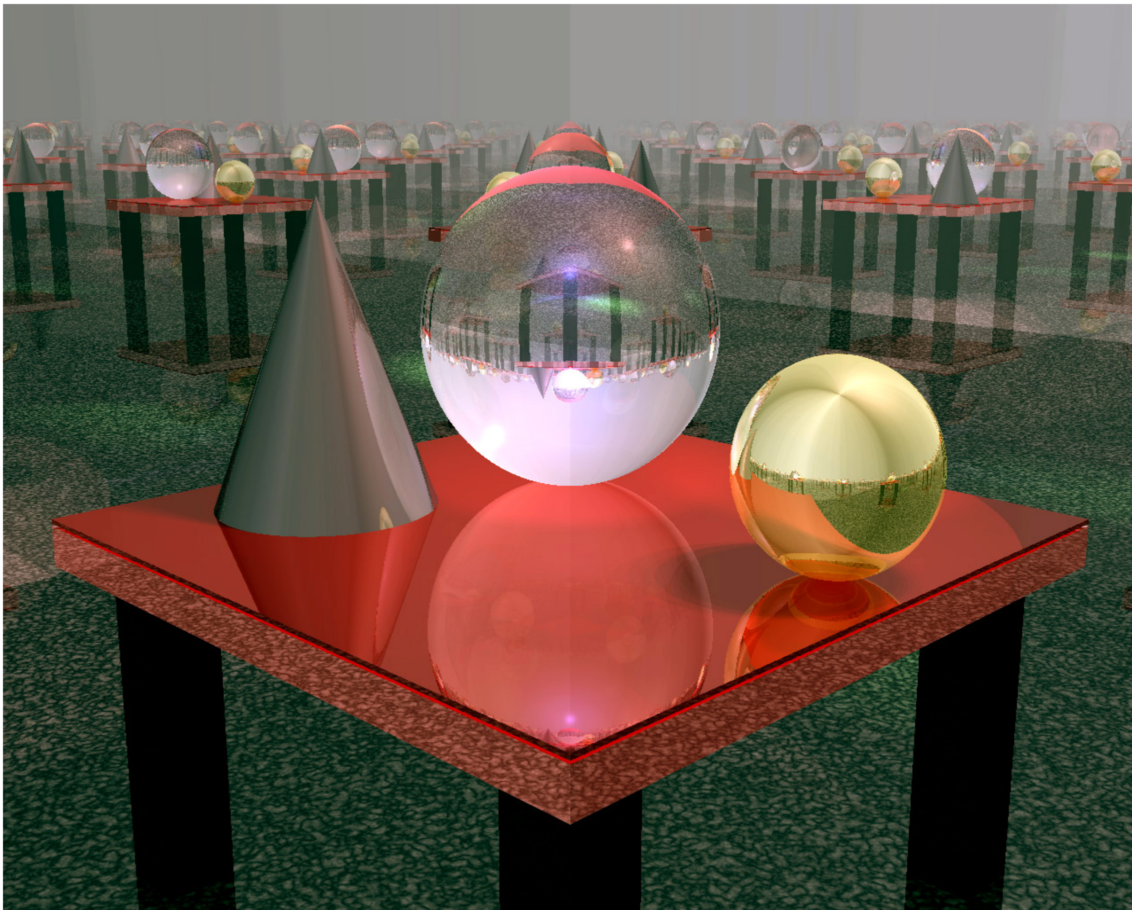


Abbildung 7.2. Die Szene aus Abbildung 7.1 wurde in *Povray* nachgestellt. Weichere Schatten, Korrekturen am Brechungsindex und Photon Mapping sind bereits integriert.

durch Strahlverfolgung miteinander verrechnet. Das Programm *POVRAY* (*P*ersistence *O*f *V*ision *RAY* tracer) von Buck und Collins ist das zur Zeit weitverbreitetste und frei verfügbare Softwarepaket (`http://www.povray.org`), das sämtliche gängige Algorithmen enthält und mit denen versucht wird, ein photorealistisches Bild virtueller Objekte zu erzeugen.

Nachteile:

- Extremer Aufwand! Das Rendern eines einzelnen Bildes liegt je nach Genauigkeit und Anzahl der Objekte (und Geschwindigkeit des Rechners) im Minuten- (Stunden-, Tage-) Bereich.
- Bei Streulicht an diffusen Oberflächen versagt auch das Raytracing, hierzu ist ein vollständiger Strahlungstransport *(Radiosity)* zu berechnen.
- Das Bündeln von Lichtstrahlen *(Kaustik)* ist nur mit nochmals erhöhtem Aufwand möglich *(Photon Mapping, Path Tracing)*.

7.1 Raytracing arbeitet im Objektraum

Zu jedem Punkt in der Bildebene (Pixel) wird die erscheinende Farbintensität aufgrund der dort sichtbaren Szeneninformationen berechnet, indem man einen *Strahl* (ray) *vom Auge des Betrachters durch den Punkt in die Szene* bis zu einer gewissen *Schachtelungstiefe* verfolgt. Dieses klassische Verfahren wird auch als *Backward Raytracing* bezeichnet. Der Sichtstrahl summiert an jedem Schnittpunkt zunächst die (geeignet gewichtete) Lokalfarbe und dann aus geänderter Raumrichtung die Lokalfarben, die sich hier spiegeln oder die durchscheinen.

Der erste Schnittpunkt mit einem (in der Regel undurchsichtigen) Objekt gibt das sogenannte *Raycasting* oder *Hidden surface removal*.

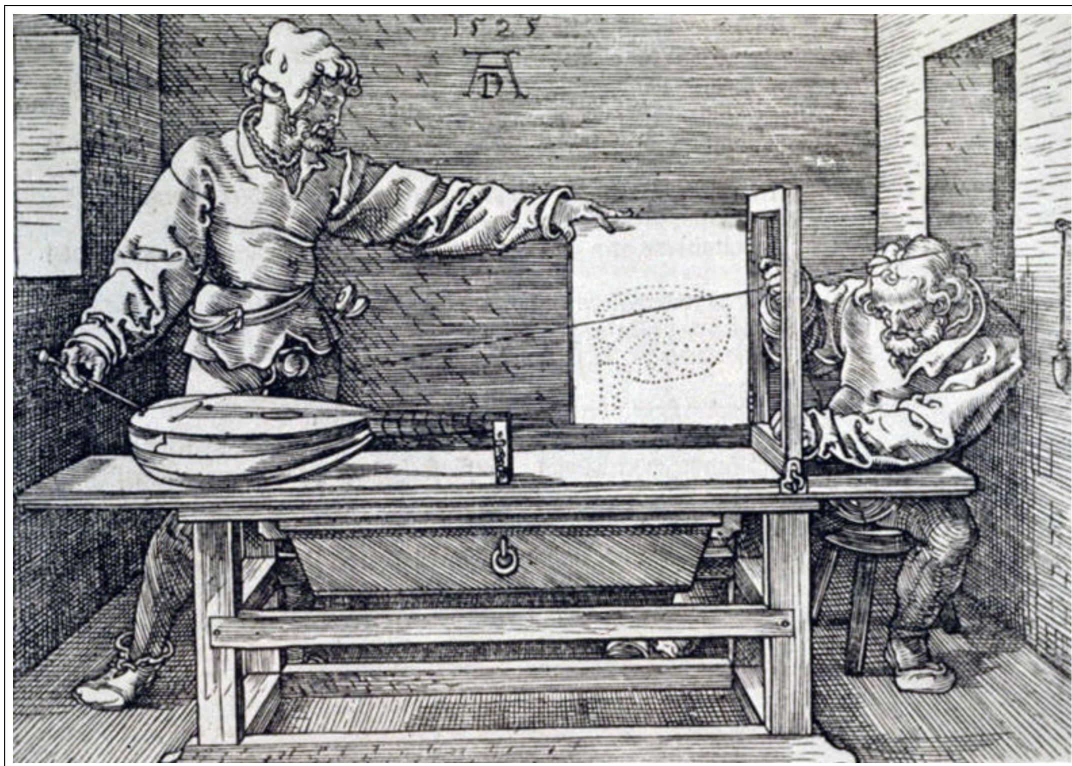


Abbildung 7.3. Bereits in der Renaissance hatte man Techniken entwickelt, um Projektionen der 3D Welt auf 2D Bildträger korrekt darzustellen (Albrecht Dürer, Mann der eine Laute zeichnet, Druckgraphik, 1525). Diesen frühen Apparaten folgte später die *Camera obscura (= dunkler Raum)*, der ein kleines Loch und darin möglichst noch eine Linse besaß. Lichtstrahlen, die von Objekten reflektiert wurden, projizierten ein Bild auf die Rückwand der Kamera.

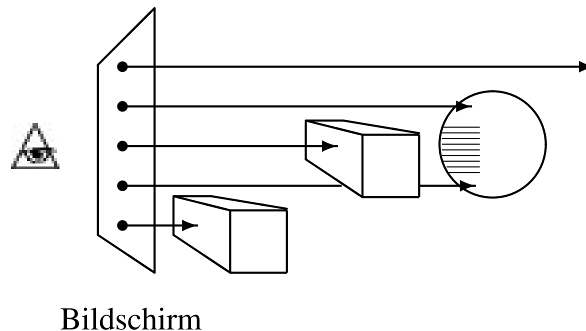


Abbildung 7.4. Das *Raycasting* im Objektraum liefert das einfache Verdecken von Objekten in Betrachterrichtung.

7.2 Historisches aus der Optik

René Descartes (1596 – 1650) behandelt in seinem Traktat von 1637 die Reflexion und Brechung von Licht in einem sphärischen Wassertropfen. Er erkannte bereits die Aufspaltung des Sonnenlichtes in Spektralfarben. Parallel in die Sphäre einfallendes Licht wird an der Grenzfläche zwischen Luft und Wasser gebeugt und reflektiert. Der maximale Winkel, unter dem ein Lichtstrahl den Tropfen in Betrachterrichtung (nach doppelter Lichtbrechung) wieder verlässt, wird daher *Descartes Winkel* genannt. Für den Brechungsindex von Wasser beträgt dieser Winkel 42°.

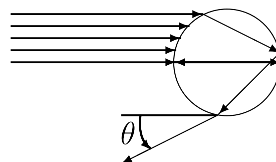


Abbildung 7.5. Der *Decartes Winkel* bei der Lichtbrechung in einem sphärischen Wassertropfen.

7.2.1 Farbaufspaltung

Isaac Newton (1625 – 1712) hat später die Abhängigkeit von der Wellenlänge bei der Lichtbrechung an Grenzflächen beschrieben, aber erst der englische Physiker George Airy (1801 – 1892) konnte die komplette Theorie durch umfangreiche Berechnungen aufstellen. Der größere Winkel von 42° gilt für langwelliges, energieärmeres rotes Licht und ein Winkel von ca. 40° für kurzwelliges, energiereicheres violettes Licht. Dadurch erklärt sich die Form des Regenbogens, der vom Sonnenlicht auf einer Wasserwand erzeugt wird: abhängig von der Position des Betrachters, der die Sonne im Rücken haben muss, erscheint der Bogen in den Farben (von außen nach innen:) Rot, Orange, Gelb, Grün,

Blau, Indigo, Violett. Der Bogen ist damit als Schnitt eines Kegelmantels mit einer senkrecht zur Kegelachse befindlichen homogenen Wasserwand aus einzelnen Wassertröpfchen zu sehen. Die Spitze des Kegels ist die jeweilige Betrachterposition, der Öffnungswinkel des Kegels liegt für die erste (Doppel-) Brechung zwischen 40° und 42°. Ein lichtschwächerer äußerer Bogen mit Farbumkehr, der durch doppelte Brechung im Wassertropfen entsteht, lässt sich manchmal bei 51° beobachten. Dass das Innere des Kegels heller erscheint, liegt an der Überlagerung aller gebrochenen und nach innen abgelenkten Spektralfarben zu Weißlicht.

Beim Phänomen des Regenbogens spielen sowohl Reflexion und Brechung als auch Interferenz und Beugung eine Rolle. Die Deutung beruht auf den Gesetzen der geometrischen Optik. Neuere Abhandlungen berücksichtigen den Teilchencharakter des Lichtes und erklären das Phänomen als Photonen-Streuung im Wassertröpfchen. Ein Sonnenstrahl tritt von einem optisch dünneren in ein etwa kugelförmiges und optisch dichteres Medium. Aufgrund der unterschiedlichen Brechungsindizes der beiden Medien ändert sich die Richtung des Lichtstrahles an der Grenzfläche, wobei diese Brechung zusätzlich wellenlängenabhängig erfolgt. Auf der Rückseite erfährt der nun in die Spektralfarben aufgespaltene Strahl eine Reflexion und wird nach den Gesetzen der geometrischen Optik zurückgeworfen. Beim Austritt aus dem dichteren in das dünnere Medium erfolgt eine weitere Brechung des Strahls.

Abbildung 7.6. Bild eines Regenbogens mit deutlich hellerem Innern.

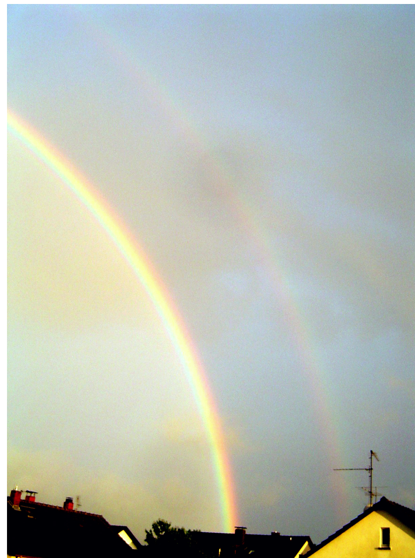


Abbildung 7.7. Nebenregenbogen mit Farbumkehr.

7.3 Implementierung: Binärer Baum und Schachtelungstiefe

Letztlich ist das *Raytracing* der natürlichste Zugang zu einer Vorstellung von Beleuchtung künstlich erzeugter Szenen. Man verfolgt einen Lichtstrahl in die Szene hinein und ändert seine Intensität und Richtung je nach der Art des Objekts, auf das er trifft. Man gibt ein Entscheidungskriterium vor, nach dem der Strahl nicht weiter verfolgt wird, meist die Schachtelungstiefe, also die Anzahl von Objektgrenzen, auf die der Strahl getroffen ist. Dadurch entsteht ein binärer Baum, der bei einer Schachtelungstiefe n die Anzahl von $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ Schnittpunkten aufweist.

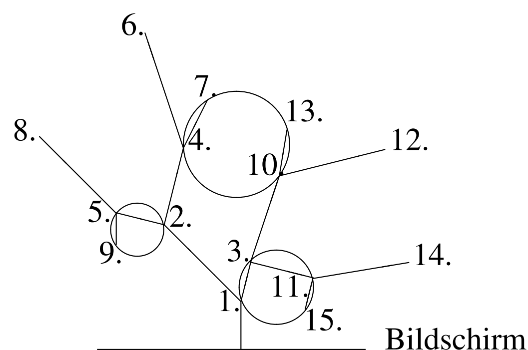


Abbildung 7.8. Drei Kugeln innerhalb einer Szene reflektieren und transmittieren einen Strahl. Dabei entsteht ein (binärer) Baum mit $2^4 - 1 = 15$ verschiedenen Schnittpunkten. Die Schnittpunkte 6, 8, 12 und 14 treffen dabei auf ein virtuelles Hintergrundobjekt.

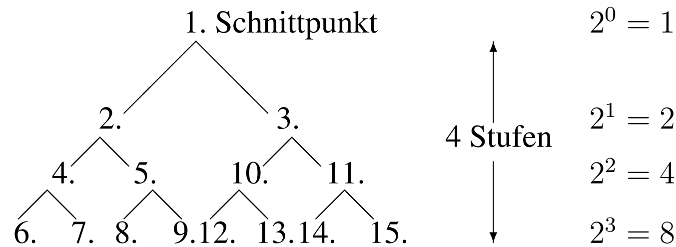


Abbildung 7.9. Diese Ereignisse entsprechen einer Schachtelungstiefe von vier.

7.3.1 Rekursiver Algorithmus

Wegen des rekursiven Aufbaus des Algorithmus wurden die Programme für Raytracing traditionell in Pascal geschrieben. Allerdings ist das rekursive Aufrufen heute genauso in C zu programmieren.

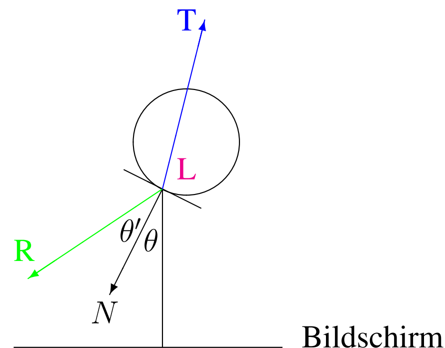


Abbildung 7.10. An jedem Schnittpunkt wird der Strahl binär aufgespalten. Die Farbe am Startpunkt des Bildschirmpixels berechnet sich aus dem lokalen Farbanteil L und den Anteilen, die vom reflektierten Strahl R und transmittierten Strahl T aus der Szene aufgesammelt werden.

Der folgende Pseudocode soll das Verfahren veranschaulichen, das Resultat der Prozedur ist die Farbe des Pixels:

```
procedure TraceRay(start, direction: vectors;
                   depth: integer; var color: colors);
var intersection_point,
    reflected_direction, transmitted_direction: vectors;
    local_color, reflected_color, transmitted_color: colors;
begin
   if depth > maxdepth then color: black
   else
   begin
```

```
      {intersect ray with all objects and find intersection point
      (if any) that is closest to start of ray}
      if {no intersection} then color:= background_color
      else
      begin
         local_color:={contribution of local color model at
         intersection point}
         {calculate direction of reflected ray}
         TraceRay{intersection_point, reflected_direction,
         depth+1, reflected_color);
         {calculate direction of transmitted ray}
         TraceRay{intersection_point, transmitted_direction,
         depth+1, transmitted_color);
         Combine(color, local_color, local_weight_for_surface,
         reflected_color, reflected_weight_for_surface,
         transmitted_color, transmitted_weight_for_surface);
      end
   end
end{TraceRay}
```

Diese Implementierung geht von der Aufteilung der Szene in endlich viele Einzelstrahlen aus, die à priori keine räumliche Kohärenz kennen. Eine Implementierung auf paralleler Hardware ist daher sehr einfach möglich.

Eine einfache Steigerung der Effizienz des Algorithmus ist durch einen schnellen Abbruch bei Erreichen der Schachtelungstiefe gegeben: Man bestimmt immer zuerst die Lokalfarbe, setzt die reflektierte und die transmittierte Farbe auf Schwarz. Ist die Schachtelungstiefe erreicht, berechnet man nur noch die Lokalfarbe und springt dann ans Ende des Programms. Eine lange Berechnung von Farbanteilen und Strahlrichtungen, die keine Verwendung mehr finden, wird dadurch vermieden.

Bemerkung 7.1 *In dem globalen Illuminationsmodell wird über die Lokalfarbe ein Streulichtterm mit einem nicht streuenden Term (binäre Aufspaltung) gemischt. Das führt je nach Wichtung zwischen lokal:global zu mehr oder weniger harten Brüchen. Der Übergang zwischen opaquen Objekten zu transparent/reflektiven Objekten ist nicht kontinuierlich.*

7.3.2 Fortschritte des Verfahrens

Um das Streulicht besser wiedergeben zu können, als es mit einer binären Aufteilung des Strahls möglich ist, wird von Robert Cook 1984 das *Diffuse Raytracing* eingeführt. Jetzt können nicht nur harte Schatten sondern auch ein weicher Verlauf berechnet werden, indem man statt einzelner Strahlen mehrere stochastisch gestörte Strahlen verfolgt und aus den berechneten Farben den Mittelwert bildet. Bildrauschen, das durch die Zufälligkeit der berechneten Farbwerte entsteht, muss anschließend wieder unterdrückt werden.

James Kajiya hat 1986 das Raytracing über seine Rendergleichung auf eine mathematisch modellierte Basis gestellt, wobei den Sekundärstrahlen eine größere Bedeutung als der Hierarchiestufe beigemessen wird. Dieses sogenannte *Path Tracing* entspricht eher den physikalischen Gesetzmäßigkeiten, weist aber einen hohen Rauschanteil bei kleinen Lichtquellen und Kaustiken auf.

Forward Raytracing, also die Verfolgung von Lichtstrahlen emittierender Lichtquellen, hat als isoliertes Verfahren wenig Sinn, da nicht gewährleistet werden kann, dass ausgehend von den Lichtquellen überhaupt alle Pixel des Schirms erreicht werden. Im *Bidirektionalen Path Tracing* allerdings wird es gezielt berücksichtigt.

Abbildung 7.11. Der Däne Henrik Wann Jensen ist heute Associate Professor in San Diego, CA.

Die letzten großen Entwicklungen fanden mitte der neunziger Jahre statt. Henrik Wann Jensen führte zur Modellierung der Kaustik (Bündelung des Lichts durch Brechung und dadurch entstehende helle Lichtflecken) das *Photonmapping* mit einer speziellen Technik zur Speicherung von Partikelstrahlen als Ergänzung in das Raytracing ein. Eric Veach und Leonidas Guibas erweiterten das Path Tracing zum sogenannten *Metropolis Light Transport (MLT)*, indem sie Pfadkombinationen, über die viel Energie transportiert wird, dauerhaft speichern. Die Informatik-Arbeitsgruppe von Philipp Slussalek an der Universität Saarbrücken arbeitet aktuell erfolgreich an Chipsätzen für Echtzeit-Raytracing, bei der die Parallelisierbarkeit des Verfahrens ausgenutzt wird.

Abbildung 7.12. Die von Henrik Jensen entwickelten Algorithmen können auch die Bündelung von Lichtstrahlen (hier die Kaustik am Fuß eines Cognacglases) recht gut wiedergeben.

7.4 Schnittpunktbestimmung

Zunächst parametrisiert man den Strahl. Zu einem Anfangspunkt (x_1, y_1, z_1) beschafft man sich einen weiteren Punkt (x_2, y_2, z_2) auf dem Strahl und erhält eine Geradengleichung für $t > 0$ als

$$\begin{aligned} x &= x_1 + (x_2 - x_1)t &= x_1 + it \\ y &= y_1 + (y_2 - y_1)t &= y_1 + jt \\ z &= z_1 + (z_2 - z_1)t &= z_1 + kt. \end{aligned}$$

7.4.1 Schnittpunkte mit Kugeln

Sehr eindrucksvolle Bilder wurden mit spiegelnden und semitransparenten Kugeln erzeugt, die die Bilder einer umgebenden Szene verzerrt oder durch Brechung und Spiegelung auf den Kopf gestellt reflektieren.

Die S^2 Sphäre mit Radius r und Mittelpunkt (l, m, n) genügt der Gleichung

$$(x-l)^2 + (y-m)^2 + (z-n)^2 = r^2.$$

Um die Schnittpunkte eines Strahls mit sphärischen Objekten zu berechnen, setzt man den parametrisierten Strahl in die Kugelgleichung ein und löst die eine quadratische Gleichung $at^2 + bt + c = 0$ mit Koeffizienten

$$\begin{aligned} a &= i^2 + j^2 + k^2 \\ b &= 2i(x_1 - l) + 2j(y_1 - m) + 2k(z_1 - n) \\ c &= l^2 + m^2 + n^2 + x_1^2 + y_1^2 + z_1^2 - 2(lx_1 + my_1 + nz_1) - r^2. \end{aligned}$$

Als Entscheidungskriterium, ob ein Schnittpunkt vorliegt, benützt man üblicherweise die Diskriminante D, die man zur Lösung der Gleichung jedenfalls berechnen muss:

$$D = \left(\frac{b}{a}\right)^2 - \frac{4c}{a}, \qquad t_{1,2} = \frac{1}{2}\left(-\frac{b}{a} \pm \sqrt{D}\right).$$

Ist $D < 0$, so schneidet der Strahl die Kugel nicht (die Lösungen sind imaginär). Für $D = 0$ liegt der Strahl tangential zur Kugel. Nur für $D > 0$ muss für den reflektierten und transmittierten Strahl jeweils Richtung und Farbanteil bestimmt werden. Die Reihenfolge der Schnittpunkte durch n Kugeln auf dem Strahl wird durch eine aufsteigende Folge von $t_i,\ i = 1, ..., 2n$ bestimmt.

Ein weiterer Vorteil von sphärischen Objekten besteht in der leichten Bestimmung der Normalen: Die Oberflächennormale im Schnittpunkt ist der normierte Vektor vom Zentrum der Kugel durch den Schnittpunkt.

$$((x-l)/r, (y-m)/r, (z-n)/r)^T \quad \text{ist Normale in} \quad (x, y, z)^T$$

Die Normale wird zur Berechnung des lokalen Farbanteils benötigt, wo lokale Farbmodelle für das Shading eingesetzt werden.

7.4.2 Schnittpunkte mit Polygonen

Beliebige Objekte sind in der Computergraphik meist aus Dreiecken zusammengesetzt. Dreiecke sind ebene Polygone bzw. aus drei Punkten wird eine Ebene festgelegt. Daher bestimmt man zunächst den Schnittpunkt des Strahls mit der durch drei Punkte bestimmten Ebene.

1. Schritt: Schnittpunkt des Strahls mit der Ebenengleichung $Ax + By + Cz + D = 0$

$$A(x_1 + it) + B(y_1 + jt) + C(z_1 + kt) + D = 0 \qquad \Leftrightarrow$$

$$t = \frac{Ax_1 + By_1 + Cz_1 + D}{Ai + Bj + Ck}, \qquad Ai + Bj + Ck \neq 0.$$

Sollte der Nenner Null sein, sind die Ebene und der Strahl parallel. Keine weiteren Berechnungen sind nötig.

2. Schritt: Liegt der Schnittpunkt innerhalb des Polygons?

Dazu projiziert man Polygon und Strahl auf die Ebenen, denen jeweils eine Koordinate fehlt, und testet nach der Paritätsregel. Wenn auf allen drei Ebenen (X,Y), (X,Z) und (Y,Z) der projizierte Strahl das projizierte Dreieck trifft, gibt es einen Schnittpunkt des Strahls mit dem Polygon. Die 3D Koordinaten des Schnittpunkts ergeben sich jetzt durch lineare Interpolation der projizierten Koordinaten.

Bemerkung 7.2 *Dieses Verfahren ähnelt dem z-Buffer Algorithmus. Auf dem (X,Y)-Rasterschirm erscheint nur die Projektion eines Polygons, nicht aber die z-Koordinate. Sie dient einzig als Entscheidungskriterium für die Darstellung überhaupt.*

7.5 Beschleunigung des Algorithmus

Bei einer direkten Implementierung werden in Szenen moderater Komplexität 95% der Rechenzeit für die Schnittpunktsberechnung verwendet. Daher sollte man diesen Teil des Algorithmus beschleunigen.

7.5.1 Z-Sort

Beim sogenannten *Z-Sort* werden die Strahlen derart transformiert, dass sie entlang der Z-Achse einfallen. In einem zweiten Schritt werden die Objekte in gleicher Weise transformiert und schließlich nach Z-Werten sortiert. Weiter entfernte oder überdeckte Objekte brauchen nicht näher untersucht werden. Bei diesem Verfahren interessiert immer nur der erste Schnittpunkt des Strahls mit dem Objekt. Danach ändert der (aufgeteilte) Strahl seine Richtung und der Sortieralgorithmus muss erneut durchgeführt werden.

7.5.2 Begrenzende Volumen

Es empfiehlt sich ebenfalls, für die Objekte begrenzende Volumen *(Bounding Volumes, Bounding Boxes)* anzugeben, deren Koordinaten einen schnelleren Schnittpunkttest zulassen. Kugeln oder Quader

bieten sich hier je nach Objektdimensionen an, wobei eine umhüllende Kugel im Fall von sehr langen und dünnen Objekten ungünstig ist . Viele Strahlen treffen die Kugel, ohne das Objekt zu treffen; alle diese Strahlen müssen aber im nächsten Schritt auf einen Schnittpunkt mit dem Objekt untersucht werden.

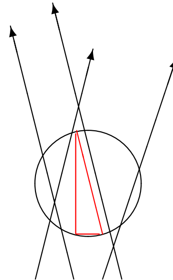


Abbildung 7.13. Kugeln als begrenzende Volumen sind bei langen, dünnen Objekten ungünstig.

Wenn man die darzustellende Szene in einem Vorverarbeitungsschritt näher untersuchen kann, ist es sinnvoll, ganze Hierarchien von begrenzenden Volumina zu speichern, die vom kleinsten zu immer größeren Objekten in Gruppen zusammengefasst werden *(bottom up)*.

7.5.3 Raumaufteilung - Fast Voxel Transversal Algorithm

Nach einem etwas anderen Prinzip verfahren die Raumaufteilungsmethoden, die auf Fujiama (1985) und Amanatides und Woo (1987) zurück gehen. Hierbei wird zunächst die gesamte Szene in einer *Bounding Box* begrenzt und das entstandene Volumen mit einem äquidistanten Gitter unterteilt. Danach werden Listen erstellt, die die Objekte einem oder mehreren dieser Kuben zuordnen. Schließlich werden Projektionen der Szene samt Gitter und Strahlen auf die Flächen der *Bounding Box* erstellt und Schnittpunkte in den nichtleeren Quadraten berechnet. Dieses Verfahren wird auch *Fast Voxel Transversal Algorithm* genannt.

7.5.4 Granulierung

Eine recht einfache Idee der lokalen Verfeinerung geht auf Arbeiten von Warnock (1969) zurück und ist als *Warnock subdivision algorithm* in der Literatur zu finden.

Es gibt genau vier Möglichkeiten (siehe Abbildung 7.15) für die Relationen eines Polygons mit einem Flächenelement: (1) umgebend, (2) schneidend, (3) enthalten, (4) disjunkt.

Legt man ein Grobgitter auf dem Rasterschirm zugrunde, muss man jedes einzelne Flächenelement auf die vier Möglichkeiten hin testen und gegebenenfalls Verfeinerungen vornehmen. Weitere Unterteilungen sind allerdings unnötig, wenn einer der folgenden Fälle zutrifft:

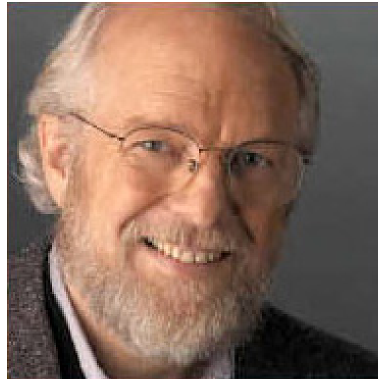


Abbildung 7.14. John Warnock gründete mit Charles Geschke die Firma Adobe.

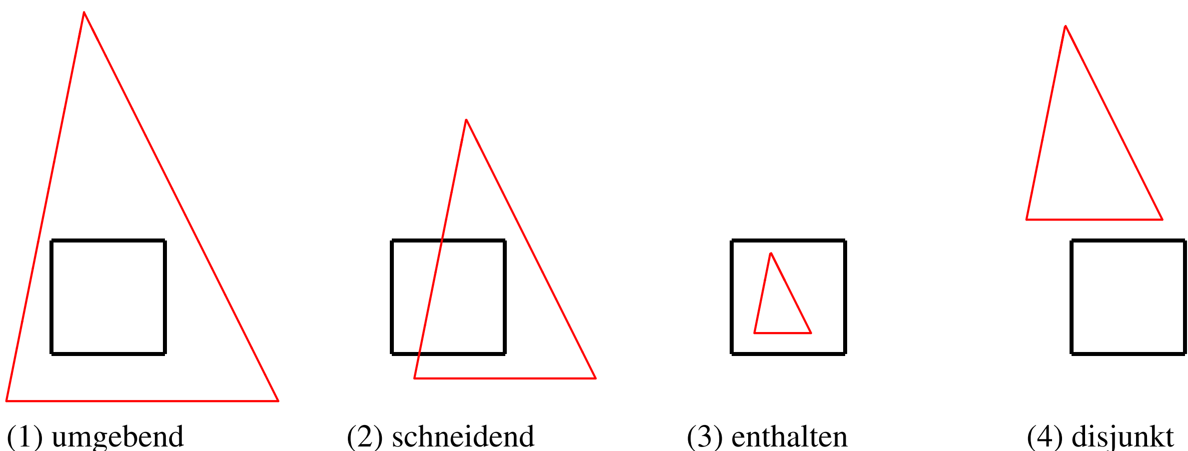


Abbildung 7.15. Es gibt vier Möglichkeiten für die Relationen eines Polygons (rotes Dreieck) mit einem Flächenelement (schwarzes Quadrat).

a. Das Flächenelement und alle Polygone sind disjunkt (4):
   $\longrightarrow$ Flächenelement bekommt Hintergrundfarbe

b. Es gibt genau ein schneidendes (2) oder enthaltenes (3) Polygon:
   $\longrightarrow$ Flächenelement bekommt Hintergrundfarbe
      $\longrightarrow$ *Scan Conversion* des Polygons und Darstellung in Polygonfarbe

c. Das Polygon umgibt das Flächenelement (1):
   $\longrightarrow$ Flächenelement bekommt Polygonfarbe

d. Viele Polygone liegen im Flächenelement, aber das im Vordergrund liegende Polygon umgibt das Flächenelement:
   $\longrightarrow$ Flächenelement bekommt Polygonfarbe

Alternativ muss man feiner unterteilen. Dieses Verfahren wird so lange betrieben, bis die Flächenelemente sämtlich eindeutig einem der Fälle a, b, c oder d zugeordnet werden können (siehe Abbildung 7.16). Diese rekursive Unterteilung geht auf Catmull (1974) zurück.

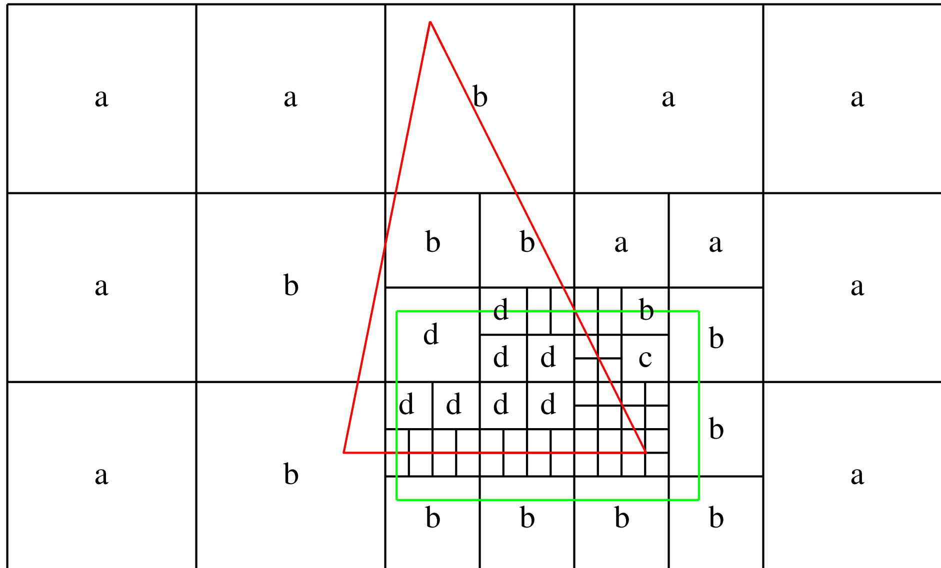


Abbildung 7.16. Bei einer adaptiven Verfeinerung muss solange weiter unterteilt werden, bis jedes Rechteck einen der Fälle a. bis d. entspricht. In diesem Beispiel wird das grüne Rechteck von einem roten Dreieck überdeckt.

7.5.5 Vergleich der Performance

Z-Werte können auf einem Rasterschirm immer nur in (möglichst realistischen) Projektionen sichtbar gemacht werden. Zum Abschluss dieses Kapitels soll deshalb ein Performance-Vergleich auf die Vor- und Nachteile der Algorithmen hinweisen, die sich mit der dritten Raumkoordinate beschäftigen.

Normiert auf einen Tiefensortier-Algorithmus (*Depth Sort*) für hundert Polygone ergibt sich folgender Vergleich:

Anzahl der Objekte	100	2.500	60.000	
Depth Sort	1	10	507	← extrem abhängig
Z-Buffer	54	54	54	← (fast) völlig unabhängig
Scan Line	5	21	100	
Warnock Subdivision	11	64	307	

Tabelle 7.1. Quelle: Foley et al.,[FvDF+96b], p. 473.

Der Vergleich zeigt eindrücklich, wie sehr der Tiefensortier-Algorithmus von der Anzahl der Objekte abhängig ist. Solange relativ wenige Objekte nach ihren jeweiligen Z-Werten sortiert werden, ist dieser Algorithmus gegenüber einem Z-Buffer-Algorithmus im Vorteil, der jedenfalls alle Pixel einmal auf ihren zugehörigen Z-Wert hin überprüfen muss. Da die Anzahl der Pixel konstant bleibt, auch wenn die Zahl der Polygone wächst, muss immer nur ein Farbwert genau ermittelt werden, während alle Farbwerte von verdeckten Objekten gar nicht berechnet werden.

7.6 Übungsaufgaben

Aufgabe 7.1 Brechungsindex

Schreiben Sie ein einfaches Raytracing-Programm, mit dem Sie eine semitransparente Kugel vor einem schachbrettartigen Hintergrund darstellen. Das Schachbrett nimmt von oben nach unten in jeder Reihe immer intensivere Farben an. Variieren Sie den Brechungsindex ihrer Kugel von einem optisch dichteren zu einem optisch dünneren Medium als dem umgebenden. Was beobachten Sie?

Aufgabe 7.2 Transparenz und Opazität

Zeichnen Sie mehrere Reihen von gleichgroßen Kugeln vor einen schachbrettartigen Hintergrund, der deutlich kleiner gemustert ist, als der Kugelradius. Beginnen Sie mit einer völlig opaquen, diffus reflektierenden Kugel und steigern Sie gleichmäßig von Reihe zu Reihe die Transparenz und in jeder Spalte die Reflexionseigenschaften. Wo ist der Unterschied zwischen aufeinanderfolgenden Kugeln am größten?

Aufgabe 7.3 Schachtelungstiefe

a) Zeichnen Sie drei spiegelnde Kugeln unterschiedlicher Größe hintereinander, die sich zum Teil überdecken. Steigern Sie die Schachtelungstiefe dieser Szene von 1 beginnend. Ab welcher Tiefe lässt sich keine Verbesserung des Bildes mehr feststellen?

b) Verändern Sie den Kugelradius und die Fenstergröße. Welcher Einfluss ergibt sich hinsichtlich der optimalen Schachtelungstiefe?

Aufgabe 7.4 Prisma

Zeichnen Sie ein Prisma, dass das Weißlicht in das Spektrum eines Regenbogens zerlegt, indem Sie die Farbkanäle Ihres linienartig einfallenden Strahls bei der Brechung unterschiedlich handhaben.

Aufgabe 7.5 OpenGL und Povray *Visualisieren Sie eine Szene in OpenGL, die über einem Schachbrettmuster drei unterschiedlich gefärbte und geformte beleuchtete Objekte zeigt, und stellen Sie die*

gleiche Szene in Povray nach. Achten Sie insbesondere auf einheitliche Blickwinkel, Positionen, Objekte und Materialeigenschaften. Dokumentieren Sie beide Ergebnisse mit Screenshots bzw. gerenderten Povray-Bildern (wobei aus den Namen der Bilder hervorgehen soll, mit welchem Programm die Szene erstellt wurde). Rendern Sie zusätzlich ein Povray-Bild, bei dem Sie die Materialeigenschaft der Schachbrettebene um einen Reflexionskoeffizienten erweitern und wählen Sie eine Kameraposition, bei der man die Spiegelungen der Objekte auf der Oberfläche sieht.

Anhang A

DIN A-Formate und 150 dpi-Umrechnung

1	inch	$=$	25.4	mm
1	cm	$\approx$	0.3937	inch

DIN A0			84.1	cm	$\cdot$	118.9	cm	$=$	10000	cm^2
	3 Byte Farbtiefe	$\cdot$	4966	dots	$\cdot$	7023	dots	$\approx$	105	MB
DIN A1			59.5	cm	$\cdot$	84.1	cm	$=$	5000	cm^2
	3 Byte Farbtiefe	$\cdot$	3511	dots	$\cdot$	4966	dots	$\approx$	52	MB
DIN A2			42.0	cm	$\cdot$	59.5	cm	$=$	2500	cm^2
	3 Byte Farbtiefe	$\cdot$	2483	dots	$\cdot$	3511	dots	$\approx$	26	MB
DIN A3			29.7	cm	$\cdot$	42.0	cm	$=$	1250	cm^2
	3 Byte Farbtiefe	$\cdot$	1756	dots	$\cdot$	2483	dots	$\approx$	13	MB
DIN A4			21.0	cm	$\cdot$	29.7	cm	$=$	625	cm^2
	3 Byte Farbtiefe	$\cdot$	1241	dots	$\cdot$	1756	dots	$\approx$	6.5	MB
DIN A5			14.9	cm	$\cdot$	21.0	cm	$=$	312.5	cm^2
	3 Byte Farbtiefe	$\cdot$	878	dots	$\cdot$	1241	dots	$\approx$	3.3	MB
DIN A6			10.5	cm	$\cdot$	14.9	cm	$=$	156.25	cm^2
	3 Byte Farbtiefe	$\cdot$	620	dots	$\cdot$	878	dots	$\approx$	1.6	MB

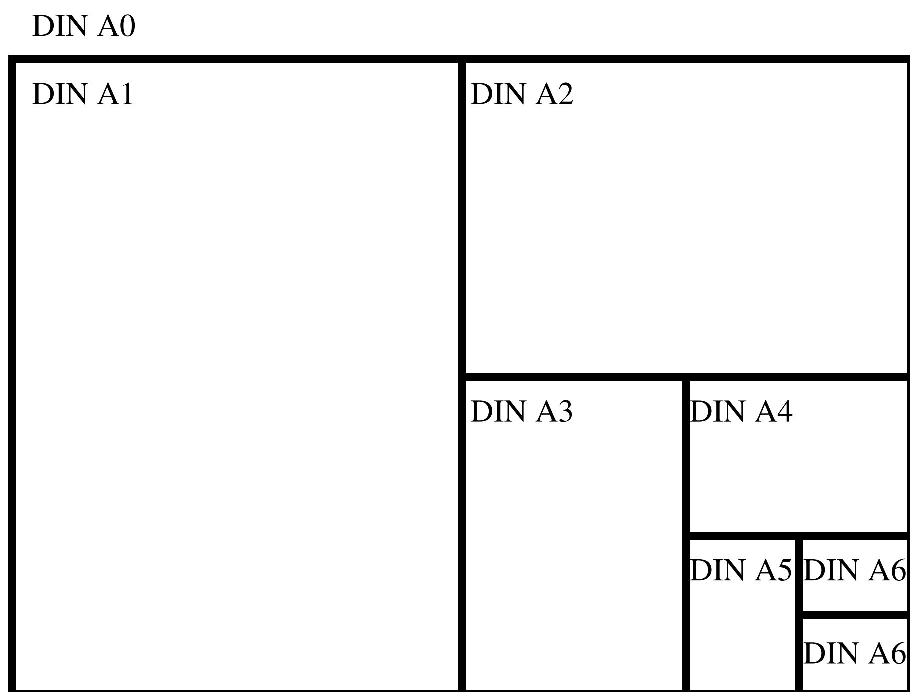


Abbildung A.1. Die Halbierung der längeren Seite lässt das jeweils kleiner Format mit jeweils größerer Ziffer entstehen.

Literaturverzeichnis

[Ebe01] EBERLY, D. H.: *3D game engine design: a practical approach to real-time computer graphics*. Academic Press, Morgan Kaufmann Publishers, 2001.

[FvDF+96a] FOLEY, J. D., A. VAN DAM, S. K. FEINER, J. F. HUGHES und R. L. PHILLIPS: *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990, 1994, 1996.

[FvDF+96b] FOLEY, J. D., A. VAN DAM, S. K. FEINER, J. F. HUGHES und R. L. PHILLIPS: *Introduction to Computer Graphics*. Addison-Wesley, 1990, 1994, 1996.

[Kil96] KILGARD, MARK J.: *OpenGL Programming Guide for the X-Windowsystem*. Addison-Wesley, 1996.

[SM00] SCHUMANN, H. und W. MÜLLER: *Visualisierung: Grundlagen und allgemeine Methoden*. Springer-Verlag, Berlin, Heidelberg, 2000.

[SWND04a] SHREINER, DAVE, MASON WOO, JACKIE NEIDER und TOM DAVIS: *OpenGL programming guide: the official guide to learning OpenGL, version 1.4*. OpenGL Architecture Review Board, Addison-Wesley, 2004.

[SWND04b] SHREINER, DAVE, MASON WOO, JACKIE NEIDER und TOM DAVIS: *OpenGL reference manual: the official reference document to OpenGL, version 1.4*. OpenGL Architecture Review Board, Addison-Wesley, 2004.

[Wat90] WATT, A. H.: *Fundamentals of three-dimensional computer graphics*. Addison-Wesley, 1989, 1990.