New APIs for Mobile Graphics

Kari Pulli

Nokia Research Center and MIT CSAIL, Cambridge, MA, USA

ABSTRACT

Progress in mobile graphics technology during the last five years has been swift, and it has followed a similar path as on PCs: early proprietary software engines running on integer hardware paved the way to standards that provide a roadmap for graphics hardware acceleration. In this overview we cover five recent standards for 3D and 2D vector graphics for mobile devices. OpenGL ES is a low-level API for 3D graphics, meant for applications written in C or C++. M3G (JSR 184) is a high-level 3D API for mobile Java that can be implemented on top of OpenGL ES. Collada is a content interchange format and API that allows combining digital content creation tools and exporting the results to different run-time systems, including OpenGL ES and M3G. Two new 2D vector graphics APIs reflect the relations of OpenGL ES and M3G: OpenVG is a low-level API for C/C++ that can be used as a building block for a high-level mobile Java API JSR 226.

Keywords: Mobile Graphics, Graphics APIs

1. INTRODUCTION

Few years ago, at the turn of the millennium, interactive mobile graphics sounded like a contradiction in terms. Mobile devices, such as cellular phones, had very small monochrome displays, slow CPUs, and little memory. Today, several APIs exist both for 2D and 3D graphics, there are hardware designs to accelerate their execution, and the features of mobile graphics engines begin to rival those of desktop systems. The change has been very quick.

There are several reasons why mobile devices have more limited resources than are available on desktop. The most fundamental reason is that mobile devices have to obtain the power from batteries. As the amount of power is limited, also the performance of the system must be limited lest the battery runs out too quickly. Also, the small size of the devices limits how much power can be consumed, even if better batteries were available. Liquid cooling or fans are not practical, so great care must be taken in thermal design to avoid hot spots that might burn the whole chip. Finally, most mobile devices are mass-market consumer products, for which customers are typically not ready to pay as much as they might for a high-performance engineering or even home entertainment workstation.

Yet the resources have improved. The CPU development has followed Moore's law[1] yielding smaller and higher performance CPUs. Similarly, the amount of available memory has increased. Perhaps the most important enabler for graphics, however, has been the fast improvement of displays. That development was first fueled by the demand from digital cameras, though now the greatest demand probably comes from mobile phones. A typical mobile phone around year 2000 had a 84×48-pixel 1-bit monochrome display, refreshed a few times per second, but today 16-bit RGB displays are becoming the norm, with a typical display resolution around 320×240 pixels, refreshed 30 times per second.

The application that drives the graphics technology development on mobile devices most seems to be the same one as on PCs: interactive entertainment, especially gaming. According to some reports the worldwide mobile gaming market was worth two billion dollars in 2005 and it is still rapidly growing. Other entertainment applications include virtual pets and animated screen savers. Some uses do not require a 3D API, 2D vector graphics is often sufficient for animations and presentations. Both 2D and 3D graphics can be used to spice up the user interface, as well as to display mapping information.

Lots of traditional 2D graphics is bitmap graphics where the main primitive is a raster image that is directly pasted on a screen, while 3D graphics is based on vectorized shapes that are first projected to a virtual camera

Further author information: E-mail: kari.pulli@nokia.com

before displaying on the screen. Bitmap graphics works efficiently for static content at constant resolution. However, for animated bitmap graphics the storage requirements grow rapidly since each frame of animation must be stored as a separate bitmap. Also, if the same content is displayed on screens of different resolutions, the images have to be filtered, which blurs sharp patterns such as text when the image is minified and creates blocking artifacts when the image is magnified. Since mobile devices usually do not have hard drives, and the screen sizes and even orientations vary a lot, vector graphics has several advantages over bitmap graphics on mobile devices.

This article is a survey of recent open standards for mobile vector graphics. We cover OpenGL ES and M3G, two 3D APIs; Collada which is an interchange format and API for 3D graphics content; and two 2D APIs, OpenVG and JSR 226.

2. OPENGL ES

OpenGL ES[2] is based on OpenGL,[3–5] probably the most widely adopted 3D graphics API ever. OpenGL is the main 3D API on MacOS, Linux, most Unixes, and it as also available on Windows. Similarly, OpenGL ES has now been adopted as the main 3D API for Symbian OS. OpenGL itself is based on an earlier IRIS GL API[4] and has been the basis of several high-level APIs such as Open Inventor,[6] VRML,[7] and Java 3D.[8] OpenGL is a low-level API which abstracts the graphics rendering process into a graphics pipeline that closely matches many hardware implementations.

OpenGL has a sound basic architecture which potentially keeps the API compact. All features are orthogonal, meaning that it is quite easy to predict the effects of enabling or disabling a feature, and orthogonality keeps both HW and SW implementations modular. However, OpenGL was not designed to minimize the size of the API and its memory footprint, and it has evolved through several iterations. Also, it has features meant for 2D graphics such as user interfaces and presentation graphics, yet it doesn't provide a full feature set for high-quality 2D presentations. This means that in some cases the same end result could be obtained through several ways, and not all functionality has found widespread use.

The Khronos Group (www.khronos.org) took OpenGL as a starting point for a leaner version targeted for Embedded Systems, OpenGL ES.[9] The main design goals were minimizing the use of resources that are scarce on mobile devices. The API's footprint (size of the implementation) should be minimized, and expensive but rarely used functionality, as well as redundancy, should be eliminated. A detailed study on the design of OpenGL ES is available by Pulli *et al.*[10]

2.1. OpenGL ES 1.0

The first version of OpenGL ES aimed to create an API that enables a compact and efficient implementation on mobile devices such as cellular phones that have only HW support for integer arithmetics. The standardization work begun in 2002 and the specification was released in summer 2003.

OpenGL ES 1.0 pipeline begins with vertex arrays that define vertex positions, normal vectors, texture coordinates, and colors and materials. The vertices are transformed using modelview and projection matrices and lighted using the Phong lighting model. The vertices are combined into point, line, or triangle-based primitives, which are rasterized, that is vertex data is interpolated for each pixel within the primitive. A color from texture map can be combined with the triangle color, several tests such as depth, alpha, stencil, and scissor tests can eliminate the pixel, and finally the pixel color can be blended with the earlier color on that pixel.

The first design target, compactness, was mostly achieved by eliminating features from OpenGL 1.3.[3] However, some features are more basic than others, and more difficult to emulate using the remaining features or application code than others. Features in the back end of the graphics pipeline belong to such category, especially on systems with HW rasterization support that provide no direct access to the frame buffer. Therefore OpenGL ES retained all the fragment operations such as blending modes and logic operations. Texture mapping API was slightly simplified without sacrificing its expressive power. 1D and 3D textures were eliminated, but 2D textures (that can trivially emulate 1D textures) were retained. Desktop OpenGL had different ways to define texture coordinates for single and multitexturing, OpenGL ES unified the entry points and kept only the multitexturing version, though implementations do not have to provide more than one texture unit.

On the other hand, features that are in the front end of the pipeline can be easily emulated in application code. Such functionality includes the GL Utility library, spline evaluators, feedback and selection modes, display lists, automated texture coordinate generation, and user clip planes. Also queries of the dynamic state of the API were eliminated as an application can keep track of the state itself.

Removing redundancy makes the API more compact. All of OpenGL's 2D operations can be emulated through the remaining 3D functionality, hence most of them were removed. For example line stippling and drawpixels (pasting a 2D pixel rectangle on screen) can be emulated through texture mapping. The original OpenGL 1.0 way of defining the primitives to be drawn consisted of a begin-statement that starts the primitive, followed by an arbitrary sequence of changes to the current value of a vertex position, normal, texture coordinates, or color, possibly mixed with application code, followed by an end-statement. This makes a quite complicated state machine. OpenGL 1.1 introduced vertex arrays, which combined all the vertex information into a few arrays which can be passed to the graphics API in a single function call. As this enables both a simpler and faster implementation, vertex arrays became the only way of defining primitives in OpenGL ES. Also the set of primitives was simplified. Only point, line, and triangle-based primitives were kept, while quadrangles and general polygons were eliminated. Quads and polygons are typically internally split into triangles for rendering in any case, and in case of non-planar polygons, are not uniquely defined.

OpenGL uses heavily floating-point numbers, but a typical hand-held device does not have hardware support, and emulating IEEE floats[11] on integer-only hardware can be quite slow. This problem was tackled with several approaches. For example, all double precision floats were eliminated from the API and replaced with single precision floats. Additionally, a new data type was created, `glFixed`, which is a 32-bit fixed-point number consisting of a signed 16-bit integer part followed by a 16-bit decimal part. For every function with floating-point arguments, a variant was created that takes in fixed-point arguments. The main profile, called Common Profile, retains all OpenGL accuracy requirements for matrix operations, hence makes the profile easy to use. For very limited systems, the Common-Lite Profile eliminates all floating-point functions from the API and relaxes the range and accuracy requirements of matrix operations, making the profile even more compact, but also more difficult to use since the matrix operations can quite easily overflow.

There is also a third profile, called Safety Critical Profile. Since that is meant for environments such as car and aviation displays and medical devices, where expensive safety certification is required but more hardware resources are typically available, the profile is fairly different from the "mobile" Common Profile and is not described here.

2.2. OpenGL ES 1.1

OpenGL ES 1.1 was completed one year after the first version. Whereas 1.0 was as minimalist an API as possible, 1.1 targets systems that have a bit more resources, including possibly specialized graphics hardware.

Seven new features were introduced, five of them mandatory and two optional. Possibly the one making the biggest difference in visual quality is better support for texture mapping. Now it is required that the implementations support at least two texturing units, making multitexturing possible. Also dot3 texture mapping is available, allowing rendering objects consisting of relatively few polygons, yet appearing as if they had very detailed geometry.

Texture maps were encapsulated in texture objects already in 1.0, allowing caching texture data on the graphics engine, providing faster execution and lower power consumption. In 1.1 also vertex data can be encapsulated in a vertex buffer object, providing similar performance benefits.

Point sprites allow defining a texture map that can be projected with a single vertex (as opposed to using two triangles and four vertices), rendered as a screen-aligned rectangle, and attenuated in size based on the distance from the camera. This provides efficient support for particle effects such as fire, smoke, or rain. A user clip plane is now available, useful for example for portal culling, and dynamic state can be queried, which makes it easier to create reusable software components that can store the API state, perform its own rendering, and restore the state, without the application having to explicitly keep track of the global state.

The two optional extensions are draw texture, which allows using the texturing machinery to copy screen-aligned rectangles to the frame buffer, and matrix palette, which supports linear blend skinning useful in animating articulated objects such as human characters.

Figure 1. Example of OpenGL ES 1.1 content.

2.3. OpenGL ES 2.0

In OpenGL ES 1.x, the graphics pipeline has fixed functionality. That is, the algorithm of each pipeline stage is fixed, though most stages can be parameterized (e.g., choosing the blending mode) or be disabled so the data just by-passes that stage unprocessed. OpenGL 2.0 defines the OpenGL Shading Language,[12] which allows two parts of the graphics pipeline run a program called a *shader*. The *vertex shader* replaces the vertex transformation (both modeling and projection) and lighting stages of the fixed-functionality pipeline. Each vertex is processed separately, and the triangle connectivity cannot be changed, but by associating other data such as the positions of the neighboring vertices arbitrary, even non-linear combinations and transformations can be applied to the vertex. Also, an arbitrary lighting model can be applied that may either more closely approximate physical light-matter interaction than the Phong lighting model used in OpenGL and OpenGL ES 1.x does, or produce non-photorealistic shading models such as cartoon rendering. The vertex shader is followed by a stage that interpolates data associated with vertices, and for each fragment (a sample of a pixel) within a triangle (or line or point) a *fragment shader* is executed. For example, for high-quality lighting the lighting calculations can be performed separately for each pixel rather than at vertices. The fragment shader can access interpolated vertex data, as well as previous values written into that fragment, but cannot access the values of neighboring fragments (though rendering to texture and multipass algorithms can overcome even that limitation). Before the values are committed to the frame buffer, a few additional fixed-functionality stages are applied to the fragment (various tests, color buffer blend, and dithering).

OpenGL ES 2.0 adopted the OpenGL shading language with few modifications. However, while desktop OpenGL decided to keep all the old fixed-functionality entry points, OpenGL ES simplified the design by eliminating all the functionality that shaders replace (all matrix and lighting operations for vertex shader, texture mapping, color sum, and fog for fragment shader). Additionally the pipeline was simplified by eliminating user clip planes and logic operations. Removing the redundant fixed functionality enables a more compact implementation of the API, as well as makes it simpler to use. Though the 2.0 API is not backwards compatible with 1.x, the API was designed so that hardware manufacturers can provide both 1.x and 2.0 drivers for the same hardware, allowing 1.x application to be executed on the same device.

2.4. EGL

EGL[2] is an API originally designed to support and standardize the integration of OpenGL ES to the software platform or operating system. It is similar to GLX[13, 14] and abstracts handling of resources such as frame buffers and window. EGL can provide OpenGL ES with three types of rendering surfaces: windows (on-screen rendering

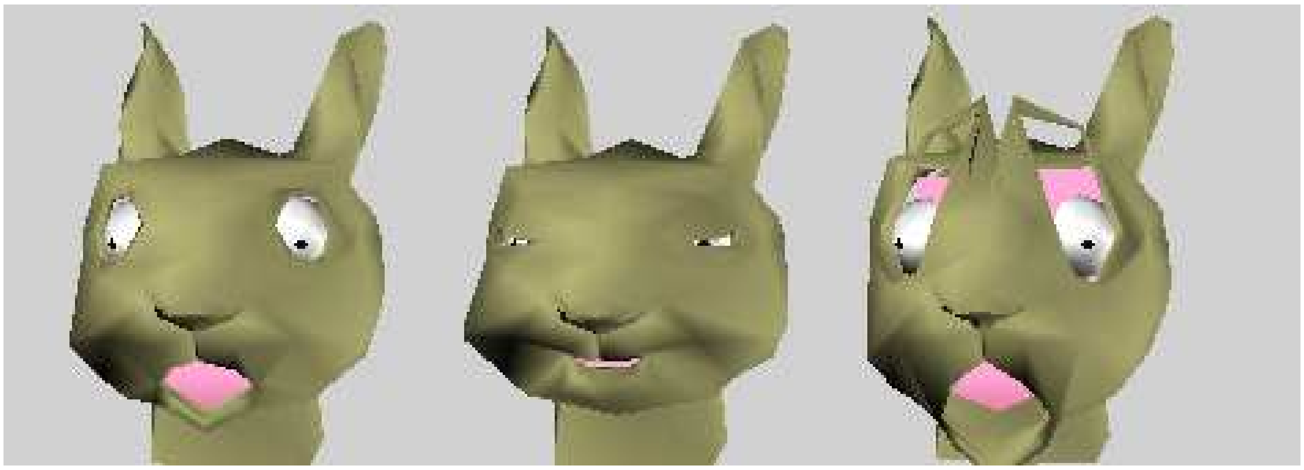


Figure 2. M3G has extensive animation support, including mesh blending that allows interpolation and exaggeration based on a few target meshes.

to graphics memory), pbuffers (off-screen rendering to user memory), and pixmaps (off-screen rendering to OS native images). Also the rendering context (the abstract OpenGL ES state machine) is allocated through EGL, and different rendering contexts may share data such as texture or vertex buffer objects.

Also OpenVG (see Section 5) has adopted EGL, and EGL has been amended so that image data can be shared between OpenGL ES and OpenVG contexts. It is possible that EGL evolves into a generic system integration API that can be adopted by other Khronos APIs, such as OpenMAX that deals with imaging and video.

3. M3G

OpenGL ES is a low-level API aimed for C or C++. Even though there are some mobile platforms such as Symbian OS and embedded Linux that allow end users to install new C/C++ applications, and the number of such devices is growing, that number is still small compared to devices that support mobile Java (or J2ME, Java 2 Micro Edition). Therefore a 3D API for J2ME is needed as well.

One option would be to create a thin wrapper that gives the programmer almost a direct access to OpenGL ES, and such a wrapper is being defined in JSR 239 (JSR = Java Standardization Request). However, that standard is not complete yet, and it is unclear how widely it will be adopted.

M3G, or Mobile 3D Graphics API for Java (JSR 184), is a high-level API. The key reason for choosing a high-level API over a low-level one is that mobile Java is much, up to 10-20 times slower than carefully written C or assembly code.[10] A typical application needs to do many other tasks than rendering, for example object and camera transformations, object database or scene graph traversal, keyframe animation, and mesh morphing and skinning. If the 3D rendering is a relatively small portion of the total workload, hardware-accelerating the rendering portion will not significantly accelerate the complete application. However, if a high-level API includes many of the other common tasks described above, the API can be implemented in fast C or assembly code, leaving only the control logic to Java.

One possible approach in defining M3G would have been to do as was done with OpenGL ES: take an existing API and make a more compact version of it. A natural candidate was Java 3D.[8] However, a simple subset of the API was not feasible, changes and extensions would have been needed. Instead, a completely new API, which borrowed many design principles from Java 3D, was created

The API defines a set of nodes that represent components such as Camera, Light, and renderable objects such as Sprite3D and Mesh. The renderable objects have an associated Appearance object, which again can have components such as Material (colors for shading), CompositingMode (defines blending, depth buffering, alpha testing, and color masking), PolygonMode (polygon culling and winding, shading type, perspective correction hint), Fog (fades colors based on distance), and Texture2D (texture blending and filtering modes, texture transformation, and Image2D containing the texture map). Sprite3D is a 2D image with a position in 3D space and can be used for billboards, labels, or icons, while Mesh represents 3D objects. A Mesh consists of a VertexBuffer defining positions, normals, colors, and texture coordinates, and a set of IndexBuffers which connect the vertices to triangle strips. Each IndexBuffer also has a matching Appearance object. There are also two special types of Meshes. A SkinnedMesh can be used to animate an articulated object, such as a human character, that has a continuous "skin". Vertices can be associated with a set of bones, and a weighted combination of bone transformations is applied to animate the vertex locations. A MorphingMesh is useful for animating unarticulated objects, and is often used for facial animation. It takes in a collection of target meshes with the same connectivity, and the output is a linear combination of the target meshes, allowing both interpolation and extrapolation of the targets.

The nodes can be connected to a scene graph, rooted by a World node. A Group node facilitates hierarchical modeling, and each node has a 3D transformation as an inherent component. Each node must have a unique parent, which means that the same object cannot appear twice in the scene graph. However, the actual data, such as vertex or index buffers can be shared, enabling memory savings through instantiation. Other scene graph functionality includes finding objects with a matching UserID tag and finding the closest object that intersects a given 3D ray.

Animation is a key part of interactive 3D content, and M3G has extensive support for animation. In addition to the SkinnedMesh and MorphingMesh classes, just about every property can be keyframe animated, and there are several ways of interpolating the keyframes. The API also includes a file format that allows all static API structures be encoded into binary files and loaded up. The existence of a file format facilitates separation of the graphics content from application logic, allowing the artists and programmers to work separately but making it easy to combine their work. The combination of a scene graph, animation support, and file loader makes it also possible to create a simple animation player in about ten lines of code.

The API allows the programmer to render the whole scene graph at once (retained mode) as well as rendering a branch or individual node at a time (immediate mode). The scene graph rendering does not create side effects such as animation, the user has to explicitly call for *animate* to update the scene graph animations, and *align* that applies auto-alignments, e.g., for billboards, before calling *render* to draw the scene graph.

The rendering model is the same as in OpenGL ES. The two standards were defined concurrently, partly by the same people. It would be feasible that a single mobile device would support both APIs, and the same basic graphics engine, especially if hardware accelerated, should be able to support both APIs. In fact, a recommended way to implement M3G is to implement it on top of OpenGL ES.

4. COLLADA

The APIs enable the execution of interactive 3D content, but creating such content can be a complicated and expensive task. Creating interesting content merely through programming is usually feasible only for toy examples and technology demonstrations. Design of interesting 3D scenes, attractive objects and characters within scenes, and their interaction requires specialized tools. A large collection of commercial tools exists for creating the geometry of models, their appearance including material properties and texture maps, and their animations. Another set of tools organize that data and massage it into a format suitable for interactive rendering through APIs such as OpenGL ES or M3G. Finally, tools called effects or FX are used to control the rendering, especially in case of multipass rendering techniques. The problem is that these tool chains are very restrictive. For example, editing data with one set of tools probably makes it impossible to modify aspects of the model with another tool and then continue editing with the first tool. Importing data to other tools not included in the highly tailored tool chain may be impossible. Also, most tools exist only for APIs available for desktop or game consoles, and creating a new suite of tools for the new mobile APIs is an expensive and slow undertaking.

Collada is a set of documents (which can be stored as files, or in a database) and an API that addresses these issues. It is an interchange format for 3D assets that retains all information of the object. It is glue that binds together various content creation tools and allows applications to talk with each other using the same language. It also allows exporting the data in a format that is suitable for different environments, making it possible to

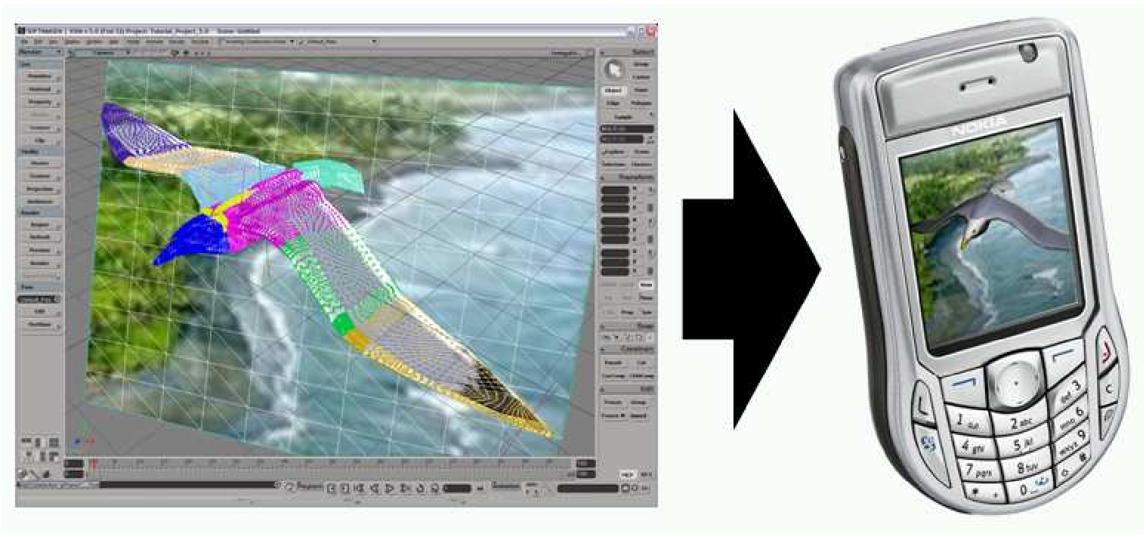


Figure 3.
Collada facilitates creation of interactive content with digital content creation tools and exporting it to a variety of systems, in this case to an M3G file displayed on a mobile phone.

create mobile content with tools originally created for desktop of consoles, as well as supporting creation of content that should run on multiple types of devices. Collada started as an open source project but has now been included into the Khronos standard family.

Collada data is stored in structured XML documents. Collada includes a set of libraries for things such as camera, geometry, and so on, and it is mainly extended by adding new types of library. The data items are called assets, such as geometry that may include arbitrary associated data in addition to the object shape and connectivity. Objects can be combined into a scene graph, and there is support for animation and skinning. The latest version (1.4, the first version since Collada joined Khronos) supports physics for basic rigid body objects that have joints. It also supports FX, which encode multi-pass rendering effects. FX handles the selection of appropriate state changes and data needed for each rendering pass, including the selection and linking of shader programs if the API supports shaders. Finally, version 1.4 includes support for OpenGL ES and M3G content.

5. OPENVG

Original OpenGL is not a pure 3D API, it contains several 2D elements. For example lines have width that is defined in the screen space instead of in 3D, and the lines can be stippled or dashed and the dashing is defined in screen space as well. During creation of OpenGL ES 1.0 there was discussion whether OpenGL ES should either be expanded to have more complete 2D functionality so that OpenGL ES could be used to implement an SVG (Scalable Vector Graphics) player, but in the end OpenGL ES remained an almost pure 3D API. The new basic primitive required by SVG, SVG-Tiny, Macromedia Flash, and the like, is an arbitrary polygon made of smooth Bezier patches and straight line segments. While uniquely defined in 2D, there is no straightforward and general extension to 3D for a polygon made out of non-planar Bezier segments.

Yet there is a clear need for high-quality 2D vector graphics functionality. Lots of digital content is inherently 2D, such as diagrams and maps (although 3D versions can sometimes be useful). Having a primitive that is smooth (Bezier) rather than approximating smoothness by combining linear segments (lines, triangles) makes for a more economical representation and higher quality images, regardless of the level of magnification. A 2D API makes it easier to incorporate many elements such as dashing, end caps, and line join styles, and other features that do not necessarily have a natural mapping into 3D.

OpenVG is a new low-level API for high-quality 2D presentation graphics. The structure of the API is quite similar to OpenGL ES so application developers familiar with OpenGL ES should be able to learn OpenVG

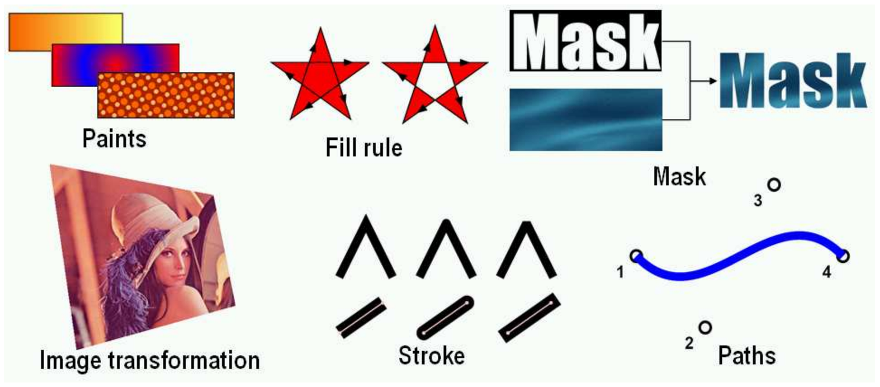


Figure 4. Examples of OpenVG elements. Paints include both linear and radial gradients and patterns in addition to simple solid colors. Various fill rules can be used to define the inside / outside areas of polygons. Masks can be used to select a shape out of an image. Images can be transformed using a general planer projective mapping, which enablers viewing images in apparent perspective. Strokes have several end caps and joints. Paths allow using Bezier curves as their components.

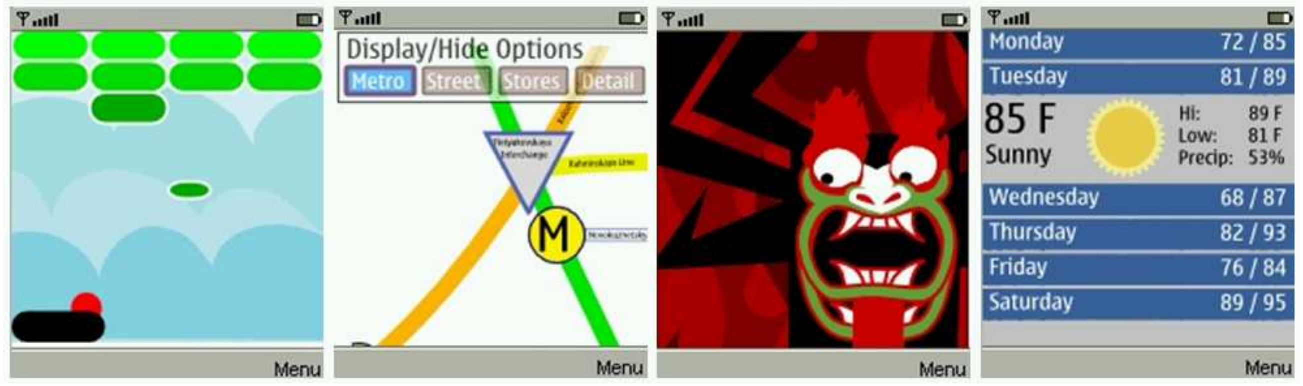
quickly. The target applications for which the API was developed include SVG viewers, portable mapping applications, e-book readers, games, scalable user interfaces, and even low-level graphics device interfaces (GDI) of mobile operating systems. Like in OpenGL ES, the use of system resources is abstracted through the use of EGL, and OpenVG and OpenGL ES may interact with each other by sharing image data and rendering into the same drawing surface.

OpenVG can be described using a similar pipeline as is familiar from OpenGL. The OpenVG pipeline consists of eight main stages. In OpenVG, all the shapes are described using paths consisting of Bezier or line segments. In the first stage the paths are created. The second stage creates a new path corresponding to the stroke geometry if the path is to be stroked, that is, if it is to get an outline. The strokes are parameterized by end cap and line join styles, line width, and dashing, among other things. The third stage transforms the paths using an arbitrary affine transformation. The fourth stage rasterizes the continuous paths, that is, determines which pixels are covered by the shapes. If a pixel is only partially covered by a shape, antialiasing can create a partial coverage value. In the fifth stage the pixels are clipped and masked to remain within drawing surface and application-given scissor rectangles and alpha masks. The sixth stage creates a paint for the remaining pixels. In addition to a simple color, the paint can be either a linear or radial gradient, or even an arbitrary pattern of shapes. The seventh stage is applied if a raster image is being rendered, in which case a planar projection is applied to the input image, and the image is sampled and combined with paint color and alpha values according to the current image drawing mode. The final stage combines the resulting color, alpha, and antialiasing information with the information already in the destination surface according to the current blending rules. Figure 4 illustrates some OpenVG features.

6. JSR 226

In the previous section we mentioned SVG players as one of the target applications for OpenVG. SVG (Scalable Vector Graphics), a W3C recommendation for vector graphics, is a language for describing two-dimensional graphics and graphical applications in XML. Some new browsers such as Firefox include now SVG support, whereas for some others a plug-in is required. SVG includes also a DOM (Document Object Model) which allows efficient vector graphics animation via ECMAScript or SMIL. However, SVG documents are mostly designed for scripted animations or presentations that can react to events, controlled by the SVG player, rather than producing complex stand-alone applications. SVG-Tiny is a profile of SVG that is designed to be suitable for mobile phones.

JSR 226 (Scalable 2D Vector Graphics) is a J2ME API that encapsulates SVG Tiny 1.1 functionality. It provides support for the XML / SVG Micro-DOM, and the Java programming language replaces the need for



Monday 72 / 85
Tuesday 81 / 89
85 F
Sunny
Hi: 89 F
Low: 81 F
Precip: 53%
Wednesday 68 / 87
Thursday 82 / 93
Friday 76 / 84
Saturday 89 / 95

Figure 5.
Examples of content generated using JSR 226. A 2D game, an interactive map with variable levels of detail, an animated cartoon, the weather forecast for the next six days.

an additional scripting language. JSR 226 provides two different modes. The first mode is a straightforward playback of SVG animation content, as shown in the third image of Figure 5. The second one gives the full control to the application, which can selectively build up an SVG description and render and modify it arbitrarily under Java program control, as demonstrated in the first, second, and fourth images in Figure 5. JSR 226 provides scalability and easy customization to J2ME applications, for example the game example (first image in Figure 5) is designed so that it automatically adapts to different screen sizes and shapes, and can change its appearance through a set of downloadable skins. The API allows creation of programmable, dynamic SVG content that reacts to user interaction, real-time network data such as traffic or weather information, and updates to location-based information.

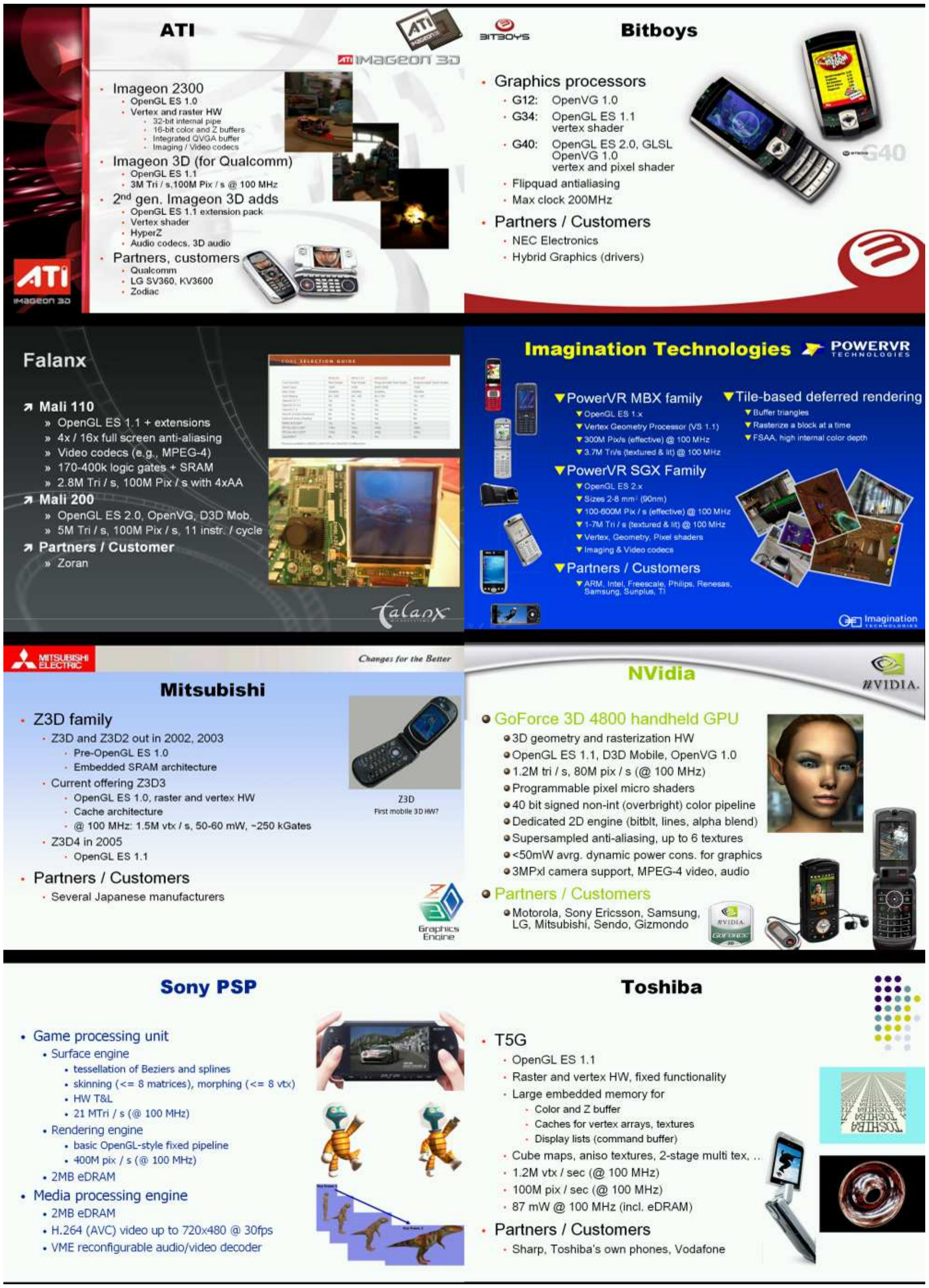
7. CURRENT STATUS

7.1. Hardware Designs

Graphics APIs place great demands for the processing power of mobile devices. Though software implementations are possible, a dedicated hardware acceleration provides better performance. But there is also another reason for specialized hardware, namely power consumption. For a given performance level, it is more power-efficient to use dedicated logic rather than run the algorithm on a general processor. That would, however, assume that the CPU can idle at least part of the time. Games often use all the available resources, thus trading performance to power.

On desktop the main concern in graphics hardware design has been usually the performance at the cost of power consumption, but in mobile designs the priorities are reversed. However, some approaches are useful to tackle both of these goals at the same time. Hardware implementations use techniques such as tile-based rendering[15–17] and texture compression[18–20] to eliminate external memory accesses (and hence power consumption) at the cost of extra on-chip logic. Other work on mobile graphics include Akenine-Möller and Ström's work on anti-aliasing, texture filtering, and occlusion culling that are suitable for low-power graphics hardware[21] and Aila *et al.*'s work on delay streams for hardware-assisted occlusion culling.[22] Kameyama *et al.* describe a very low-power accelerator for geometry processing and rasterization that has been used in some mobile phones in Japan.[23]

There is already a large offering of mobile graphics hardware from even larger number of vendors than on desktop. Figure 6 shows an overview of such designs from ATI, Bitboys, Falanx, Imagination Technologies, Mitsubishi, NVidia, Sony, and Toshiba. All of them can support OpenGL ES 1.1 level of functionality, therefore they also accelerate M3G. Many manufacturers have already announced designs supporting OpenGL ES 2.0 level programmable shaders. A typical design can rasterize one pixel per clock and support 1-3 million triangles per second.



ATI

IMAGEON 3D

- Imageon 2300
  - OpenGL ES 1.0
  - Vertex and raster HW
    - 32-bit internal pipe
    - 16-bit color and Z buffers
    - Integrated QVGA buffer
    - Imaging / Video codecs
- Imageon 3D (for Qualcomm)
  - OpenGL ES 1.1
  - 3M Tri / s,100M Pix / s @ 100 MHz
- 2nd gen. Imageon 3D adds
  - OpenGL ES 1.1 extension pack
  - Vertex shader
  - HyperZ
  - Audio codecs, 3D audio
- Partners, customers
  - Qualcomm
  - LG SV360, KV3600
  - Zodiac

Bitboys

- Graphics processors
  - G12: OpenVG 1.0
  - G34: OpenGL ES 1.1 vertex shader
  - G40: OpenGL ES 2.0, GLSL OpenVG 1.0 vertex and pixel shader
  - Flipquad antialiasing
  - Max clock 200MHz
- Partners / Customers
  - NEC Electronics
  - Hybrid Graphics (drivers)

Falanx

- Mali 110
  - OpenGL ES 1.1 + extensions
  - 4x / 16x full screen anti-aliasing
  - Video codecs (e.g., MPEG-4)
  - 170-400k logic gates + SRAM
  - 2.8M Tri / s, 100M Pix / s with 4xAA
- Mali 200
  - OpenGL ES 2.0, OpenVG, D3D Mob.
  - 5M Tri / s, 100M Pix / s, 11 instr. / cycle
- Partners / Customer
  - Zoran

Imagination Technologies

- PowerVR MBX family
  - OpenGL ES 1.x
  - Vertex Geometry Processor (VS 1.1)
  - 300M Pix/s (effective) @ 100 MHz
  - 3.7M Tri/s (textured & lit) @ 100 MHz
- PowerVR SGX Family
  - OpenGL ES 2.x
  - Sizes 2-8 mm² (90nm)
  - 100-600M Pix / s (effective) @ 100 MHz
  - 1-7M Tri / s (textured & lit) @ 100 MHz
  - Vertex, Geometry, Pixel shaders
  - Imaging & Video codecs
- Partners / Customers
  - ARM, Intel, Freescale, Philips, Renesas, Samsung, Sunplus, TI
- Tile-based deferred rendering
  - Buffer triangles
  - Rasterize a block at a time
  - FSAA, high internal color depth

Mitsubishi

Changes for the Better

- Z3D family
  - Z3D and Z3D2 out in 2002, 2003
    - Pre-OpenGL ES 1.0
    - Embedded SRAM architecture
  - Current offering Z3D3
    - OpenGL ES 1.0, raster and vertex HW
    - Cache architecture
    - @ 100 MHz: 1.5M vtx / s, 50-60 mW, ~250 kGates
  - Z3D4 in 2005
    - OpenGL ES 1.1
- Partners / Customers
  - Several Japanese manufacturers

Z3D
First mobile 3D HW?

NVidia

- GoForce 3D 4800 handheld GPU
  - 3D geometry and rasterization HW
  - OpenGL ES 1.1, D3D Mobile, OpenVG 1.0
  - 1.2M tri / s, 80M pix / s (@ 100 MHz)
  - Programmable pixel micro shaders
  - 40 bit signed non-int (overbright) color pipeline
  - Dedicated 2D engine (bitblt, lines, alpha blend)
  - Supersampled anti-aliasing, up to 6 textures
  - <50mW avrg. dynamic power cons. for graphics
  - 3MPxl camera support, MPEG-4 video, audio
- Partners / Customers
  - Motorola, Sony Ericsson, Samsung, LG, Mitsubishi, Sendo, Gizmondo

Sony PSP

- Game processing unit
  - Surface engine
    - tessellation of Beziers and splines
    - skinning (<= 8 matrices), morphing (<= 8 vtx)
    - HW T&L
    - 21 MTri / s (@ 100 MHz)
  - Rendering engine
    - basic OpenGL-style fixed pipeline
    - 400M pix / s (@ 100 MHz)
  - 2MB eDRAM
- Media processing engine
  - 2MB eDRAM
  - H.264 (AVC) video up to 720x480 @ 30fps
  - VME reconfigurable audio/video decoder

Toshiba

- T5G
  - OpenGL ES 1.1
  - Raster and vertex HW, fixed functionality
  - Large embedded memory for
    - Color and Z buffer
    - Caches for vertex arrays, textures
    - Display lists (command buffer)
  - Cube maps, aniso textures, 2-stage multi tex, ..
  - 1.2M vtx / sec (@ 100 MHz)
  - 100M pix / sec (@ 100 MHz)
  - 87 mW @ 100 MHz (incl. eDRAM)
- Partners / Customers
  - Sharp, Toshiba's own phones, Vodafone

Figure 6.
A selection of mobile graphics hardware designs. The performance numbers have been scaled to assume a clock rate of 100 MHz.

7.2. Status Of Each Standard

This section covers a snapshot of the current status of the new standards as of January 2006.

The first mobile phones supporting OpenGL ES shipped in late 2004. Both OpenGL ES 1.0 and 1.1 ship by now, mostly in software implementations, but several hardware implementations are already shipping. An open source implementation of OpenGL ES 1.1 exists at `http://ogl-es.sourceforge.net/`. OpenGL ES 2.0 specification exists, but the final ratification requires two independent implementations that pass conformance tests, which is unlikely to happen before 2007.

M3G has proved a very popular API and can be already found in most new Java-enabled handsets. The first implementations shipped in late 2004, and by now over 100 handset models has M3G support. Especially in the Far East several operators used to provide a proprietary 3D API for mobile Java, but now it seems M3G has for the most part replaced them. In 2005 a minor update, M3G 1.1 was completed, it contained a collection of minor clarifications and bug fixes. Work for M3G 2.0 may begin in 2006, that would probably incorporate some programmable functionality of OpenGL ES 2.0.

Collada started as an open source project, but joined Khronos group in 2005. The first version that was approved by Khronos is version 1.4, it was approved in January 2006. The new features include support for shader effects, physics, and features of OpenGL ES and M3G.

OpenVG has quite similar situation as OpenGL ES 2.0. Both specifications were announced at SIGGRAPH 2005, hardware manufacturers have announced designs with API support, but no commercial implementations ship so far. The work toward OpenVG 1.1 has begun, it is likely to include better support for high quality scalable text, among other things.

The specification of JSR 226 was approved in late 2004 and the final version released in early 2005. At least two manufacturers have announced products (Nokia 6280, 6282; Motorola i870) with JSR 226 support, but none are available at the time of writing. Both JSR 226 and M3G (JSR 184) are going to be universally available since they are included as required components in JSR 248, Mobile Service Architecture for CLDC, which consolidates and aligns API specifications available on mobile devices. The work for the successor of JSR 226, JSR 287, is just starting, the expert group is being formed. Features that it targets include better support to create and modify animations, support of SVG Mobile 1.2 features, content streaming, and immediate-mode rendering that is compatible with OpenVG.

8. CONCLUSIONS

The young field of mobile computer graphics has matured at a very fast speed. The first mobile graphics engines were available in 2000 and 2001, for a few years proprietary graphics engines reigned, but starting in 2004 standard APIs and engines begun to appear and replace the proprietary engines. The latest APIs such as OpenGL ES 2.0 and OpenVG already match the graphics features available on desktop.

Until early 1980's mostly only researchers could access devices that could create interactive computer graphics. Then graphics workstations made interactive graphics available to thousands of engineers, and home computers such as Commodore 64 and Amiga did the same for even a larger number of hobbyists. In 1990's PCs replaced graphics workstations and made high quality graphics available in most homes. Now interactive graphics is available on the first truly ubiquitous computing platform, the mobile phone. Ubiquitous visual multimedia is becoming reality. The next big problem is how to create and distribute content to all these devices, tools such as Collada should help in that task.

APPENDIX A. LINKS TO SPECIFICATIONS

This article has mentioned many specifications, they are all available online. Table 1 collects the links in one location.

Table 1. Specifications and their locations on the web.

Collada	`collada.org/`
Java 3D	`java.sun.com/products/java-media/3D/`
JSR 226: Scalable 2D Vector Graphics API	`www.forum.nokia.com/java/jsr184#jsr226`
JSR 248: Mobile Service Architecture for CLDC	`www.jcp.org/en/jsr/detail?id=248`
JSR 287: Scalable 2D Vector Graphics API 2.0	`www.jcp.org/en/jsr/detail?id=287`
OpenGL, GLX, OpenGL SL	`www.opengl.org/documentation/spec.html`
OpenGL ES, EGL	`www.khronos.org/opengles/spec.html`
OpenVG	`www.khronos.org/openvg/spec.html`
M3G (JSR 184)	`www.forum.nokia.com/java/jsr184#jsr184`
SVG (1.0, 1.1, 1.2)	`www.w3.org/Graphics/SVG/`
SVG-Tiny 1.0	`www.w3.org/TR/SVGMobile`
SVG-Tiny 1.1	`www.w3.org/TR/SVGMobile11`
SVG-Tiny 1.2	`www.w3.org/TR/SVGMobile12`

ACKNOWLEDGMENTS

I'd like to thank members of the various standardization expert groups for productive collaboration that have resulted in these new APIs. Especially I'd like to acknowledge my colleagues from Nokia that I've most closely collaborated in graphics standardization: Tomi Aarnio, Tolga Capin, Suresh Chitturi, Jyri Huopaniemi, Koichi Mori, Tero Pihlajakoski, Kimmo Roimela, Jani Vaarala, and Stephen Wilkinson.

REFERENCES

1. G. E. Moore, "Cramming more components onto integrated circuits," *Electronics* **38**, pp. 114–117, Apr. 1965.
2. Khronos, *OpenGL ES Native Platform Graphics Interface (Version 1.0).* The Khronos Group, 2003.
3. M. Segal and K. Akeley, *The OpenGL Graphics System: A specification (Version 1.3).* Silicon Graphics, Inc., 2001.
4. M. Segal and K. Akeley, "The design of the OpenGL graphics interface." Silicon Graphics technical report, 1994.
5. M. J. Kilgard, "Realizing OpenGL: two implementations of one architecture," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pp. 45–55, ACM Press, 1997.
6. P. S. Strauss and R. Carey, "An object-oriented 3D graphics toolkit," in *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pp. 341–349, ACM Press, 1992.
7. ISO/IEC, *Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language (VRML) – Part 1: Functional specification and UTF-8 encoding.* International Organization for Standardization / International Electrotechnical Commission, 1997. ISO/IEC 14772-1:1997.
8. H. Sowizral, K. Rushforth, and M. Deering, *The Java 3D API Specification with Cdrom*, Addison-Wesley Longman Publishing Co., Inc., 2000.
9. Khronos, *OpenGL ES 1.0 Common/Common-Lite Profile Specification.* The Khronos Group, 2003.
10. K. Pulli, T. Aarnio, K. Roimela, and J. Vaarala, "Designing graphics programming interfaces for mobile devices," *IEEE Computer Graphics and Applications* **25**(8), pp. 66–75, 2005.
11. IEEE, *IEEE Standard for Binary Floating Point Arithmetic.* IEEE Computer Society, New York, std 754-1985 ed., 1985.
12. J. Kessenich, D. Baldwin, and R. Rost, *The OpenGL Shading Language.* 3DLabs, Inc., 2003.
13. SGI, *OpenGL Graphics with the X Window System (Version 1.3).* Silicon Graphics, Inc., 1998.

14. B. Paul, "OpenGL and window system integration." ACM SIGGRAPH 1996 Course #24 Notes, August 1997.
15. S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: high-speed rendering using image composition," in *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pp. 231–240, ACM Press, 1992.
16. M. Chen, G. Stoll, H. Igehy, K. Proudfoot, and P. Hanrahan, "Simple models of the impact of overlap in bucket rendering," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pp. 105–112, ACM Press, 1998.
17. E. Hsieh, V. Pentkovski, and T. Piazza, "ZR: a 3D API transparent technology for chunk rendering," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pp. 284–291, IEEE Computer Society, 2001.
18. A. C. Beers, M. Agrawala, and N. Chaddha, "Rendering from compressed textures," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 373–378, ACM Press, 1996.
19. S. Fenney, "Texture compression using low-frequency signal modulation," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 84–91, Eurographics Association, 2003.
20. J. Ström and T. Akenine-Möller, "iPACKMAN: high-quality, low-complexity texture compression for mobile phones," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 63–70, Eurographics Association, 2005.
21. T. Akenine-Möller and J. Ström, "Graphics for the masses: A hardware rasterization architecture for mobile phones," *ACM Trans. Graph.* **22**(3), pp. 801–808, 2003.
22. T. Aila, V. Miettinen, and P. Nordlund, "Delay streams for graphics hardware," *ACM Trans. Graph.* **22**(3), pp. 792–800, 2003.
23. M. Kameyama, Y. Kato, H. Fujimoto, H. Negishi, Y. Kodama, Y. Inoue, and H. Kawai, "3D graphics LSI core for mobile phone "Z3D"," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 60–67, Eurographics Association, 2003.