PUBLISHED IN THE
SEPTEMBER/OCTOBER 1995 ISSUE OF *The X Journal*.

HARDWARE FOR ACCELERATING OPENGL®

Mark J. Kilgard *
Silicon Graphics Inc.
Revision : 1.9

July 7, 1995

The last five OpenGL® columns discussed OpenGL programming techniques and explained source code examples. This feature article is not about programming, but instead explores the ways graphics hardware can accelerate OpenGL rendering. The intent is to provide you with enough knowledge about 3D graphics hardware architecture to evaluate and select the right graphics hardware for your needs and budget.

The OpenGL graphics system is a standard programming interface for 3D graphics. OpenGL is window system independent and is available for the X Window System, Windows NT, and OS/2. Implementations for the Macintosh and Windows 95 are planned.

OpenGL is intended for interactive graphics where specialized graphics hardware can greatly improve graphics performance. Also, OpenGL is designed to be appropriate for a wide variety of graphics systems. The same OpenGL standard should achieve good performance on computers ranging from PCs to tomorrow's fastest graphics supercomputers. This is despite two or more orders of magnitude between both the expected performance and cost at the extremes of such a range of computers.

One view of OpenGL is that OpenGL is a high-level description of the functionality that an OpenGL capable graphics subsystem must implement. Within OpenGL's prescribed functionality, there are numerous approaches to implement OpenGL for various price/performance levels.

The process of rendering interactive computer graphics is a highly specialized task that benefits greatly from graphics hardware designed to support the basic operations that occur repeatedly during graphics operations. As a rule, effectively utilized graphics hardware results in better graphics performance. At the same time, adding specialized hardware to a computer system increases the cost of the system. Deciding how many dollars a graphics system should devote to hardware for accelerating graphics operations is determined by the price/performance requirements of the complete system.

At a minimum, displaying OpenGL graphics requires video display hardware connected to a *frame buffer*. A frame buffer is a bank of memory designed to hold an array of image pixels that can be continuously scanned out to video display hardware. Beyond this minimum hardware support for displaying graphics, varying degrees of additional hardware can accelerate OpenGL rendering.

The Graphics Pipeline

In simple terms, OpenGL describes a *graphics pipeline* for performing graphics operations (OpenGL commands) to affect pixels in a frame buffer. A graphics pipeline is a logical model for decomposing graphics operations. In general, all graphics systems are organized using some type of pipeline partitioning scheme. In a broader sense (not limited to OpenGL), the general graphics pipeline can be partitioned into five stages:

1. **Generation** (G), creation, acquisition, or modification of the information to be displayed and organizing this information into application data structures.

2. **Traversal** (T) of application data structures, passing on the appropriate graphics data.

3. **Transformation** (X) of the graphics data from object-space coordinates into eye-space coordinates, performing requested lighting operations, then clipping the transformed data in clip-space, and projecting the resulting coordinates into window-space.

4. **Rasterization** (R) renders window-space primitives (like points, lines, polygons) into a frame buffer. Iterating per-vertex shading calculations, texture lookups and calculations, and per-pixel operations like depth testing are performed in this stage.

*Mark graduated with B.A. in Computer Science from Rice University and is a Member of the Technical Staff at Silicon Graphics. He can be reached by electronic mail addressed to `mjk@sgi.com`

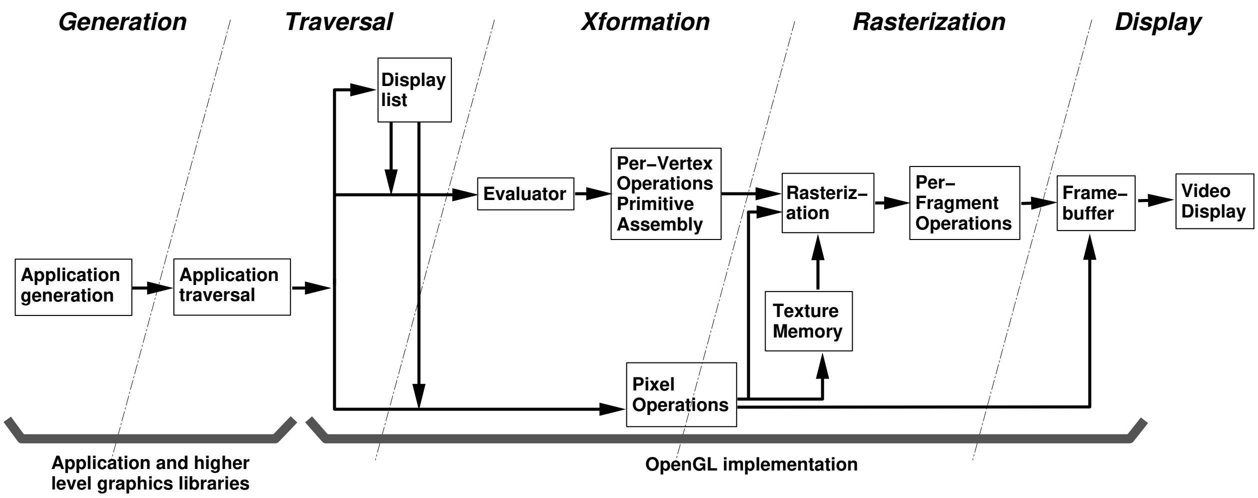

Figure 1: How the OpenGL state machine maps to the general graphics pipeline.

5. **Display** (D) scans the resulting pixels in the frame buffer, typically to a display monitor.

The stages make up a rendering pipeline because the outputs from one stage are the inputs for the next stage. The letters *G*, *T*, *X*, *R*, and *D* are used to abbreviate these stages.

An OpenGL graphics pipeline is actually a subset of the general graphics pipeline, specifying the last three stages. OpenGL leaves the generation and traversal stages to the application program since these stages are highly dependent on the particular data to be displayed and how it is organized. Tasks that are varied and highly application dependent are best left to the system's general purpose processor.[1] Figure 1 shows how the components of the OpenGL state machine map to the general graphics pipeline. Higher-level libraries like Open Inventor [11] and IRIS Performer [8] can assist applications with the generation and traversal stages.

A Taxonomy for Graphics Hardware

Using the pipeline framework, OpenGL implementations can be classified by the extent to which their OpenGL pipeline is implemented using specialized graphics hardware.

In graphics hardware architecture, a computer's general purpose processor that drives the graphics hardware is called the *host*. Deciding what stages in the graphics pipeline should be performed on the host or in specialized graphics hardware is an important question to answer when designing or evaluating a computer's graphics subsystem. Often, tasks performed on the host are said to be "implemented in software" meaning no specialized hardware is used to perform the stages. Tasks performed by specialized graphics hardware are said to be "implemented in hardware" because dedicated hardware is used to accelerate the tasks.[2]

Kurt Akeley, a coauthor of the OpenGL specification [9], has described a useful taxonomy and associated nomenclature for classifying graphics subsystems based on how the general graphics pipeline stages are partitioned between software implementation on the host and dedicated graphics hardware [1].[3] The letters `GTXRD` abbreviate the general pipeline stages. A hyphen is used to indicate what stages are (largely) implemented in hardware. For example, a PC graphics board like the standard VGA adapter would be an example of a `GTXR-D` architecture since only the display stage is implemented in hardware. Table 1 describes various graphics subsystems using Akeley's taxonomy. In practice, the taxonomy is a coarse one because there is variability in the extent to which specific functionality is implemented in hardware.

Rendering Paths

When you render a polygon using OpenGL, how the polygon is actually drawn depends on the currently enabled OpenGL modes and their associated parameters. A polygon is rendered differently depending on how lighting, texturing, blending, antialiasing, depth testing, and a host of other OpenGL modes are configured. It is useful to think of sets of rendering modes and their associated parameters as *rendering paths* through the OpenGL implementation. By enabling rendering modes and selecting their parameters, you are logically choosing a rendering path through OpenGL's state machine. The performance of actual rendering operations along a specific rendering path depends on how well that path is optimized and supported in hardware by your specific OpenGL implementation.

The `GTXRD` taxonomy is useful for obtaining a high-level,

[1] In principle, OpenGL's display list facility does support limited traversal, but OpenGL's display list functionality is not a general-purpose traversal facility.

[2] In practice, some graphics hardware does have specialized software known as *micro-code*.

[3] Akeley's original taxonomy described in [1] referred to the rasterization stage as scan conversion.

Graphics subsystem	Classification	Description
IBM VGA [4]	`GTXR-D`	standard PC “dumb frame buffer”
SGI Indigo Entry	`GTX-RD`	8-bit graphics option with line, span, and fill engine, 2-bit overlay, 4-bit double buffering
SGI Indy 24 [10]	`GTX-RD`	24-bit graphics option with line, span, and fill engine, 8-bit overlay, 12-bit double buffering
Sun Leo [3]	`GT-XRD`	210K 100 pixel triangles/sec, 4 parallel custom floating point processors and 5-way parallel raster subsystem using hyper-pipelining
SGI Indigo^2 Extreme [5]	`GT-XRD`	500K triangles/sec and 80M pixels/sec, shaded and depth buffered using 8 SIMD processors and 2-way parallel raster subsystem.
SGI Onyx RealityEngine^2 [2]	`GT-XRD`	12 MIMD i860 processors for geometry processing, highly parallel rendering subsystem, tri-linear texture mapping and real-time full-scene antialiasing
Evans & Sutherland ESIG4000	`G-TXRD`	high-end visual simulation system, includes special terrain traversal and level-of-detail rendering algorithms.

Table 1: Various graphics subsystems described using Akeley’s taxonomy for classifying graphics architectures.

coarse view of the types of operations that are potentially supported in hardware.

In practice, a system considered a `GT-XRD` architecture may incompletely support OpenGL’s full transformation and rasterization functionality in hardware. For example, a `GT-XRD` system may not have support for texture mapping in its rasterization hardware. This means any rendering path with texturing enabled will “fall back” to a software path. Falling back to software means tasks in the transformation stage that could potentially have been done in hardware without texturing enabled typically also must to fall back to software. It is important to realize that having a feature supported “in hardware” does not assure that the hardware can be used on every rendering path.

Even when rendering paths for two different sets of modes are supported via specialized hardware, you are still very likely to find that one path is faster than another. For example, when OpenGL lighting is enabled, enabling a single light is quite likely to be faster than enabling all eight possible lights. And even though depth testing is supported in hardware, a rendering path with depth testing disabled is likely to run faster than one without depth testing.

At a high-level, the `GTXRD` nomenclature is a good way to classify broadly the extent of hardware acceleration for a graphics hardware architecture. At the level of a specific application’s graphics usage, however, it is better to consider the actual rendering paths exercised by the application and decide if a given OpenGL implementation sufficiently optimizes those paths and whether the available hardware is effectively utilized.

When evaluating an OpenGL implementation for a specific application, find out what graphics functionality is implemented in hardware and what is implemented in software. Be careful not to expect every OpenGL feature to be supported in hardware. Instead, match your choice of system to your application performance demands and cost constraints. Be particularly careful about being misled by “standard” benchmarks since they may measure graphics performance for a rendering path that is not particularly important to your application. There is no substitute for your own benchmarking of the particular rendering paths you care about most.

Considering the performance of specific paths without the context of a specific application is difficult when evaluating OpenGL performance in the overall selection of a graphics workstation. Instead, let general considerations guide you like the `GTXRD` classification of the machine and general questions like whether a depth buffer is supported in hardware or what type of texture mapping hardware is provided (assuming these capabilities are broadly desirable for the system you are selecting). Benchmarks can be useful, but be wary because sometimes benchmarks are selected to show off particularly efficient rendering paths, and benchmarks more often demonstrate maximum performance as opposed to typical performance. More on benchmarks later.

The remainder of this article describes the stages of the OpenGL graphics pipeline in more detail and considers window system features for enhancing OpenGL’s window system integration. The last section takes a more global view of graphics systems and considers bottlenecks in and between different stages that can undermine overall graphics performance.

Hardware for OpenGL

If you use a PC or a low-end 2D graphics workstation, you almost certainly have graphics hardware in your system, but

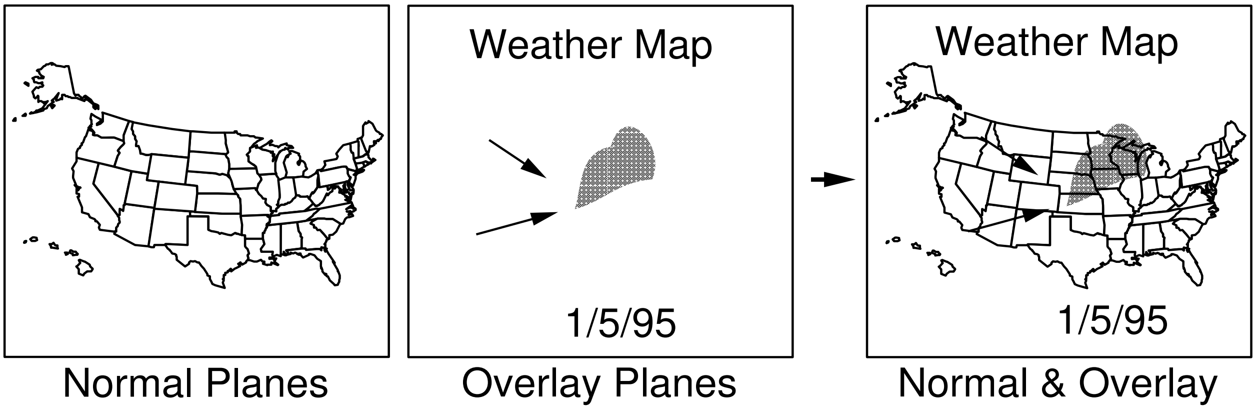

Figure 2: Overlay planes used to annotate a complex underlying image.

it is likely to be of the `GTXR-D` variety. Primarily, PC and low-end workstation 2D graphics options are evaluated on the basis of screen and color resolution and read/write bandwidth to the frame buffer. When such systems do have hardware acceleration, it is likely for simple primitives like 2D lines and rectangles. The benchmarks cited for this class of hardware are typically 2D. The 2D transformation and rasterization stages are considerably simpler than the respective stages in 3D graphics. Typically in 2D graphics, calculations are done in window coordinates so the transformation stage is sometimes nothing more than a logical placeholder.

Graphics hardware designed for 3D is more sophisticated. In addition to color values, the frame buffers for 3D hardware tend to have ancillary buffers for per-pixel data like depth and stencil values. Rasterization and often transformation tasks are off-loaded from the host processor and into the graphics hardware. To support multiple programs concurrently sending 3D graphics commands to the graphics subsystem, hardware features are often provided to better integrate 3D graphics with the window system.

The next subsections consider the graphics pipeline working in reverse order from the display stage to the transformation stage.

Display Options

OpenGL machines typically have more sophisticated display hardware compared to 2D graphics hardware. Display hardware for a 2D system typically displays the frame buffer in a uniform pixel format for every pixel on the screen.

Pixel Formats

OpenGL has two color models: RGBA[4] and color index. RGBA is very much like X's `TrueColor` visual type; color index is like X's `PseudoColor` visual type. Typically, OpenGL implementations support windows of both color models on the screen simultaneously. The "depth" or amount of color resolution for each window may also vary. So for instance, both 8-bit color index windows and 24-bit RGBA windows may co-exist on the screen at the same time.

Such display hardware needs to scan out not only the value of each pixel in the frame buffer, but also each pixel's format to determine what color to display. The X11 protocol supports multiple visuals (ie, multiple pixel formats), but most 2D low-end workstations actually support a single visual.

Double Buffering

OpenGL supports per-window double buffering. A double buffered window has two color buffers designated the *front* and the *back* buffer. The front buffer is the displayed buffer; the back buffer is used to compose the next image to be displayed. When a *buffer swap* is requested, the contents of the back buffer are logically transferred to the front buffer. Double buffering is very important for 3D since it enables smooth animation of rendered frames.

Hardware support obviates the expense of a physical data transfer and allows the buffer swap to be timed with the monitor's vertical retrace interval (when the CRT's gun is moving back to the top of the screen to begin scanning out the next frame of video). Timing the buffer swap with vertical retrace avoids a visual artifact known as *tearing* that occurs when the buffer swap occurs in the midst of the window being scanned to the monitor, getting a portion of the image from the old front buffer and a portion from the new front buffer.

Without hardware support, the back buffer is normally allocated from host memory. In this case, a buffer swap requires a copy from host memory to the frame buffer. It is also more difficult to synchronize with vertical retrace. Because rendering to off-screen memory normally makes it impossible to utilize graphics hardware acceleration, double buffering hardware is typical for 3D `GTX-RD` and `GT-XRD` systems.

Normally, hardware double buffering is implemented by dividing the frame buffer into two color buffers. For example, a 24-bit frame buffer when single buffered would support 12-

[4] Red, green, blue, and alpha.

bit double buffered windows. This introduces additional pixel formats, not only for 12-bit depth, also to describe which 12-bit buffer is currently considered the front buffer for specific windows.

Overlays

Overlays are another frame buffer layer that can be displayed preferentially relative to the normal frame buffer. Overlays are common for OpenGL capable hardware. Overlays can be supported as part of the X Window System [6], independent of OpenGL. OpenGL's GLX support for X supports overlays in a manner consistent with X's overlay support.

OpenGL scenes are often intricate and take time and resources to redraw, so overlays are particularly useful in conjunction with OpenGL. Pop-up and pull-down menus using the overlays can avoid damaging OpenGL windows that are expensive to redraw. Overlays can also provide a way to perform rubber banding and annotation of OpenGL windows. Figure 2 shows how overlay planes can be used for annotation.

Multiple Colormaps

Because OpenGL systems often support multiple pixel formats and different color models, display hardware suitable for OpenGL often includes support for multiple hardware colormaps. This means multiple windows using differently loaded colormaps can operate simultaneously without *colormap flashing*. Otherwise, the colormaps for some windows will not be installed, forcing those windows to be displayed with invalid colors. Be careful though, because even with multiple hardware colormaps, you can still exceed the number of physical colormaps and get colormap flashing.

Per-window Stereo

A 3D graphics system like OpenGL is complemented by display hardware that supports stereo output. Usually, this is done using a monitor with double the normal refresh rate and special goggles that rapidly switch LCD shutters between the two eyes in synchronization with the video frame rate. The display continuously scans out the left eye view, then the right eye view, synchronized with the LCD goggle shutters.

OpenGL supports a quad buffering scheme (stereo is almost always double buffered; stereo also adds left and right buffers in addition to front and back). Quad buffered stereo requires extra frame buffer memory and fast monitor refresh rates so OpenGL stereo support is not currently common. To get the stereo effect, OpenGL applications rendering to a stereo window must render both the right and left views, each from a slightly different eye point.

Rasterization

Rasterization is the process of converting primitives like lines and triangles (already converted into window coordinates by the transformation stage) into updates to pixel values in the window's drawable frame buffer region. OpenGL's rasterization stage can be divided into four sub-stages: primitive decomposition, texturing, fog, and per-fragment operations.

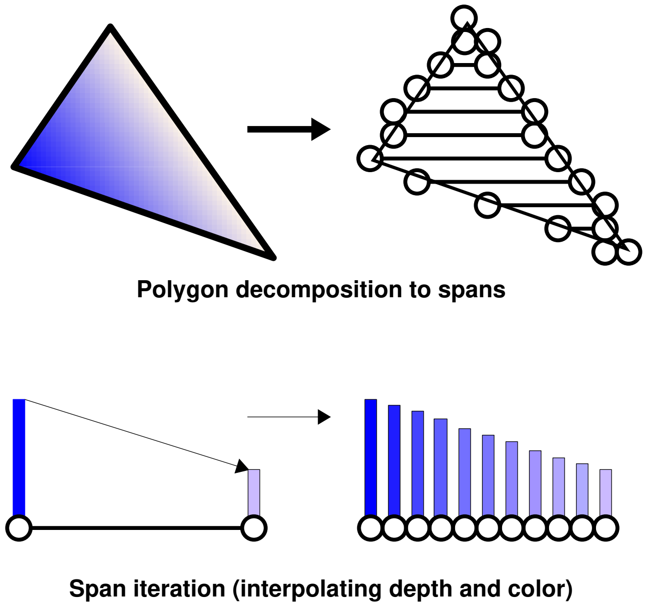

Figure 3: Example of decomposing a smooth shaded polygon to spans, then iterating across the span, interpolating depth and color between the endpoints.

Primitive Decomposition

Primitive decomposition handles geometric primitives like points, lines, and polygons as well as image primitives like pixel rectangles and bitmaps by transforming the primitives into window coordinates and determining which pixel locations are occupied by each primitive. For each occupied pixel location, a *fragment* is generated. A fragment is a pixel location accompanied by an assigned color, depth, and texture coordinates as required. The per-fragment operations that follow primitive decomposition use the fragment's associated data to update the pixel corresponding to the fragment.

OpenGL's rasterization state affects primitive decomposition. Factors affecting decomposition include antialiasing, line width, stippling, culling, and the pixel path modes.

Primitives like polygons are typically decomposed into a set of *spans*. A span is a one-pixel high horizontal line defined by a pixel location and a width. Spans can easily be converted into fragments by iterating across the span. Color, depth, and texture coordinates may be interpolated between the two span end points.

A polygon span converter can be implemented in hardware. The results from the span converter can then be fed to a hardware span engine that iterates across the generated spans doing depth and color interpolations as necessary. To lower the hardware cost of a `GTX-RD` system, the polygon span conversion may be

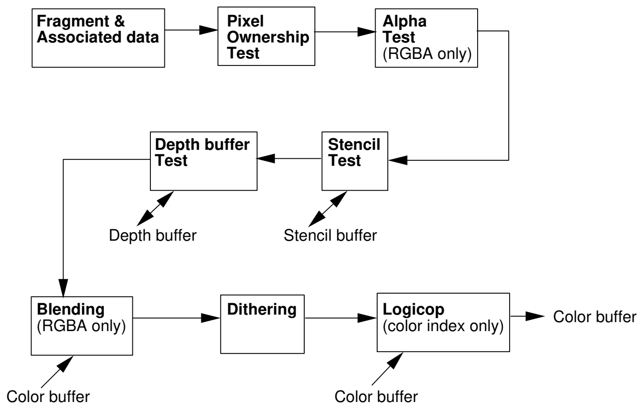

Figure 4: OpenGL per-fragment operations.

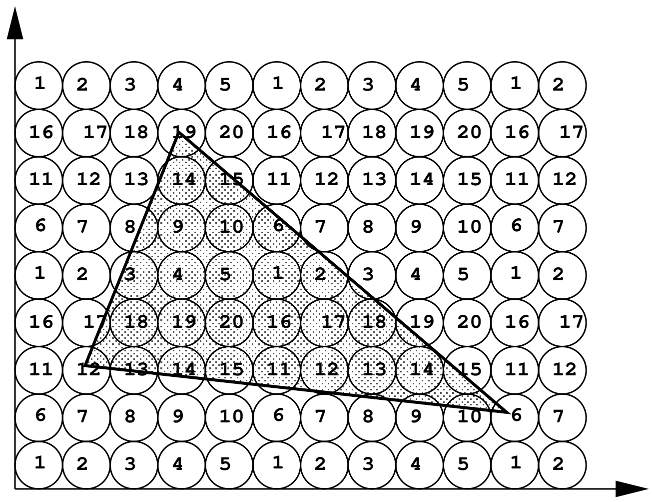

Figure 5: Example of a triangle being rasterized into a frame buffer using a 20-way image processor configuration.

done on the host so that the primitive decomposition hardware is limited to a span engine. Figure 3 demonstrates how a polygon span converter and span iteration engine combine to render a triangle into fragments. Other primitives such as points or lines can also be converted to spans, or supported by special point and line drawing engines. Similarly, so called *bit-blit* engines can efficiently render image rectangles and bitmaps. To further minimize the host overhead, graphics hardware may support Direct Memory Access (DMA) of primitives (particularly effective for image primitives). DMA lets the graphics hardware obtain data directly from host memory with minimal host overhead, thereby increasing the transfer speed to the graphics hardware.

Texturing and Fog

Between scan conversion and processing of fragment operations, the fragment color can be modified by texturing and fog. Texturing maps a portion of a specified image (called a *texture*) onto each primitive for which texturing is enabled. The effect of texturing is analogous to being able to apply wall paper over a surface. Based on the fragment's *texture coordinates*, the associated texture sample (or *texel* value) within the texture is combined with the fragment's color based on the texture function. Textures can be either 1D or 2D, and there are a host of texturing modes available in OpenGL.

Texture mapping hardware tends to be expensive because texturing is a per-pixel operation and requires fast memory to store and retrieve the texture image. Texture mapping hardware is only now becoming available on mid-range to low-end workstations.

Then, fog (if enabled) blends the fragment's post-texturing color with the current fog color based on the eye-coordinate distance and fog mode. The post-fog fragment is then used to update the fragment's associated frame buffer pixel.

Per-fragment Operations

The frame buffer in OpenGL contains more than color values. Logically, there are also *ancillary* (or helper) buffers that hold per-pixel information in the depth, stencil, accumulation, alpha, and auxiliary buffers. And as the section on the display stage discussed, there may be multiple color buffers for double buffering (left, right) and stereo (front, back). As long as the minimum GLX requirements for OpenGL's frame buffer functionality are met, an implementation may support a wide variety of frame buffer configurations with varying buffers and number of bit planes.

These ancillary buffers may be provided by the graphics hardware, usually in dedicated graphics memory. The alternative is for the host to allocate these buffers from host memory.

Some per-fragment operations affect ancillary buffers (like the depth test) while some like the scissor test do not use ancillary buffers. The per-fragment operations form a pipeline as shown in Figure 4. The operations that are most expensive are those that require an ancillary buffer read-modify-write like the depth buffer and stencil tests.

Performing per-fragment operations is particularly parallelizable since each fragment's pixel update operations can be performed independent of other updates to different pixels. An effective technique for accelerating pixel updates is replication of numerous pixel processors, each processor updating a subset of the screen's pixels. Figure 5 shows an example of how 20 pixel processors might be utilized to parallelize fragment processing.

There are cost-performance limits to replicating pixel processors. Imagine if every pixel on the screen had its *own* pixel processor. Because most pixels are not being updated at any given moment in time and many polygons tend to be small (less than 40 pixels or so), most of the screen's pixel processors would be under-utilized, so the approach would likely not be cost effective.

Operation	FLOPS
Vertex transformation	28
Normal transformation	15
Single-source lighting	28
Clip testing	6
Projection	11
Map to window space	9
Total	97

Table 2: Floating point operations required to transform a single vertex using single-source lighting.

Transforming Geometry

Vertices for OpenGL polygons, lines, and other rendering primitives are specified in object coordinates. These coordinates and other parameters like surface normals and colors are typically manipulated internally to OpenGL as floating point values. OpenGL does permit a wide variety of input formats so conversion to the OpenGL implementation's internal format may be necessary. Once converted, taking the colors, normals, vertices, and other parameters and converting these to primitives in window coordinates is the purpose of the transformation stage.

Like rasterization, transformation can be broken into several sub-stages:

Modeling transformation. Object coordinates are converted to eye coordinates using OpenGL's current modelview matrix. Surface normals are also transformed and possibly normalized. This sub-stage maintains the modelview matrix stack.

Lighting calculations. If lighting is enabled, OpenGL performs per-vertex lighting calculations based on various lighting parameters.

Projection transformation. Eye coordinates are converted to clip coordinates using OpenGL's current projection matrix.

Clip testing. Primitives are clipped to the current clip volume. Additionally, OpenGL allows specifying user-defined clip planes.

Perspective division. To maintain correct 3D perspective, OpenGL manipulates *homogenous* coordinates (meaning a 3D coordinate is represented as 4 floating point values: x, y, z, and w). This requires x, y, and z to each be divided by w to homogenize the result into a standard (x, y, z) form.

Viewport transformation. Finally, the vertex coordinates must be converted to window space based on the viewport.

These sub-stages are computationally intensive and repeated for every vertex that is transformed. The number of floating point operations required to transform a single 3D vertex lit by a single simple light source is roughly 100. See Table 2 for a breakdown. This means rendering a scene 30 times a second consisting of 10,000 triangles would require approximately 90 million floating point operations per second ($30 \times 3 \times 10,000 \times 97$) just to transform the geometry. Off-loading geometry transformation from the host processor and into specialized graphics hardware makes good sense for applications that demand very high polygon rendering rates.

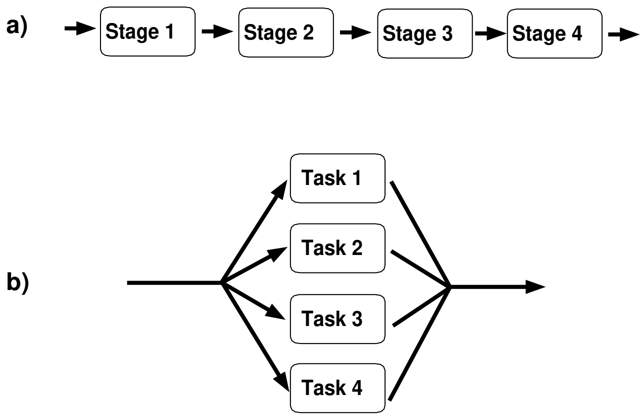

Figure 6: a) portrays a pipelined architecture with four stages; b) portrays a parallel architecture with four tasks.

Pipelining and Parallelism

Geometry transformation can be done using specialized hardware. Obtaining the required rates of floating point computation to achieve a given level of transformation performance at a reasonable cost is challenging. Two techniques are typically used to increase the throughput of transformation hardware:

Pipelining is a technique used when the stages in a task (like geometry transformation) are arranged like an assembly line. Specialized hardware performs the operations for each stage and passes the results on to the subsequent stage. Theoretically, pipelining allows a process to speed up in proportion to the number of pipeline stages added.

Parallelism is a technique used when the hardware to do a task can be replicated several times (parallel processors) and the task is divisible into largely independent units. Each unit of work is then doled out to an available hardware unit. Instead of an assembly line, think of parallelism like bank tellers servicing the queue of customers waiting for service. Theoretically, four tellers can service bank customers four times faster.

Pipelining is appropriate when a task can be broken down into stages that can be processed in linear order. Parallelism works best when the unit of work (transforming a vertex, for example) is relatively self-contained and all the units of work take similar amounts of effort. Graphics operations tend to fit both of these descriptions. Figure 6 portrays both a pipelined and a parallel architecture.

While pipelining and parallelism are discussed here in the context of geometry transformation (a particularly applicable stage in the general graphics pipeline), these two techniques can be used throughout the graphics pipeline. The example of per-fragment operations being performed by multiple image processors shown in Figure 4 is an example of parallelism. Additionally, the general graphics pipeline itself is amenable to pipelining (hence the name).

While pipelining is straightforward, it is not always as flexible as parallelism. The stages in a hardware pipeline tend to be hard-wired, meaning it is difficult to add new stages and, if the work required for any single stage grows out of proportion to the other stages, that stage can undermine the efficiency of the pipeline. For example, OpenGL supports up to eight light sources. If a single pipeline stage performs all lighting calculations, that stage might backup the pipeline. The problem could be solved by dividing the lighting stage into eight separate hardware stages. While feasible, most of the time, eight lights are not enabled so the extra lighting stages would go largely unused.

Parallelism may be more flexible than pipelining since each unit of work is executed largely independently of work executing on other parallel processors. So in the case of eight lights, each processor simply does more lighting calculations compared to the number required for a single light. Unlike a hardware pipeline, parallel processors can absorb the extra work without leaving other hardware underutilized.

Appreciate that pipelining and parallelism are not conflicting techniques, but can be complementary. For example, inside each parallel geometry processor, you are likely to find a floating point pipeline executing.

Another trade-off when designing geometry transformation hardware is whether to *hardwire* the hardware to do its specific task, or to design the hardware in a more general way so that it executes its tasks based on specialized software normally called *micro-code*. Generally, hardwired hardware can run faster, but is very inflexible, and when very complicated, can be harder to design. Hardwired hardware is more suitable for tasks that are well defined and very unlikely to change. Rasterization and display hardware is often hardwired. Usually hardware for geometry transformation is micro-coded.

Types of Parallelism

There are two common ways to implement micro-coded parallel processors: *lock-step* or *independent* processors. Independent processors let each parallel processor execute a different sequence of instructions. In the bank teller example, this might amount to one teller processing a deposit and another teller processing a withdrawal. In graphics (unlike a bank), it is often true that the last tasks are very similar to the current and immediately following tasks. For example, a triangle to be rendered is likely to be followed by several more triangles to be rendered the same way, but with different coordinates. Parallel processors can take advantage of the *homogenous* nature of graphics calculations by running in lock-step. Each graphics processor working on one of a set of triangles of the exact same type can execute the identical transformation, lighting, and clip testing micro-code, but execute the code on the coordinates of its triangle.

The advantage of lock-step execution is that a single micro-code memory store is necessary and the complexity of programming lock-step processors can be lower than programming independent processors. A single micro-code memory store is generally cheaper than having an independent micro-code store per processor. Lock-step execution works well when the workload is homogenous. If eight lock-step processors can all be working on different triangles of the same type, theoretically the triangles can be transformed eight times faster than a single processor. But, if there is an OpenGL mode change so that each triangle is not of the same type, the advantage of lock-step parallelism breaks down.

Lock-step processors are called *SIMD* processors, meaning they have a *S*ingle *I*nstruction stream for *M*ultiple *D*ata streams. Likewise, independent processors are referred to as MIMD processors, meaning they have *M*ultiple *I*nstruction streams for *M*ultiple *D*ata streams. The terms MIMD and SIMD are often used to describe the architectures of geometry transformation hardware. If other performance factors are equal (they never are!), a SIMD architecture is likely to be less expensive than a MIMD architecture, though the MIMD architecture is potentially better at less homogeneous workloads.

In a pipelined or parallel system, an N-stage pipeline or N-way parallelism does not necessarily imply a speedup of N times. When the state within a hardware pipeline changes, it is likely to require the introduction of *stalls* into the pipeline. These stalls undermine performance since all the stages are not kept busy with work. Likewise, for a parallel system, there may not be enough work to keep all N processors busy or there may be communication overhead necessary to coordinate the parallel processors.

Hardware for Window System Requirements

So far, this discussion of graphics hardware has focused on the problems of rendering and displaying graphics without concern for multiple programs trying to render graphics concurrently. Graphics workstations generally allow multiple programs to run at the same time and share access to the graphics acceleration hardware. This sharing is generally managed by the window system, the X server in the case of the X Window System.

Graphics Context Switching

The original approach that the X Window System took to sharing the graphics hardware is to allow only a single program, namely the X server, to directly manipulate the graphics hardware. X clients send rendering requests to the X server to dispatch. Effectively, the X server process acts as a rendering proxy for all X clients. For accelerated OpenGL graphics hardware, this approach has two problems.

First, OpenGL maintains a large amount of rendering state that is maintained in hardware for GTX−RD systems and even more so for GT−XRD systems. For typical 2D X operations, each rendering operation downloads its necessary hardware state since this setup overhead is small. For OpenGL, the state is potentially quite large when all the lighting parameters, matrices, modes, and texture memory is considered. For GTX−RD systems and particularly for GT−XRD systems for OpenGL to be feasible within a window system, there must be a way to efficiently context switch the graphics hardware.

Second, when accelerated graphics hardware is sufficiently fast, the overhead of generating, sending, and dispatching X protocol requests compared to the hardware's rendering speed is high. A fast graphics subsystem cannot meet its performance potential if it cannot be delivered OpenGL commands fast enough. To remedy this situation, OpenGL's GLX extension that integrates X with OpenGL provides OpenGL implementations the option of supporting *direct rendering* so that local OpenGL programs can render directly to the graphics hardware.

When the X server does all rendering to the hardware, the X server context switches the hardware as necessary. But when direct rendering is supported, graphics context switching has to be managed among unrelated processes in separate address spaces. If the graphics hardware supports context switching asynchronous to rendering, the kernel and window system can "conspire" to virtualize graphics for OpenGL programs without those programs knowing [7].

Another approach to speeding context switching is to support multiple contexts within the graphics hardware. This can be expensive because OpenGL contexts can be large and supporting multiple contexts in hardware requires hardware resources. Devoting hardware resources can be hard to justify since multiple contexts do not speed up the performance case of a single renderer. Even with multiple hardware contexts, the system may still run out of hardware contexts, so context switching OpenGL state to and from host memory must still be supported.

Fast and Arbitrary Window Clipping

In figure 4 showing the per-fragment pipeline, the Pixel Ownership Test is where fragments are thrown away if they fall outside the window's drawable region. The drawable region of a window is often called the window's *clip*.

When the X server performs 2D X rendering, the X server can perform window clipping in software based on its special knowledge of every window's clip. But in GTX−RD and GT−XRD architectures, typically the hardware must enforce window clipping because the Pixel Ownership Test occurs late in the graphics pipeline. Also when using direct rendering, OpenGL programs do not generally have knowledge of their current window clip. In these cases, window clipping is typically supported in the hardware with the hardware clipping information updated by the X server as the window layout changes.

There are two common approaches to supporting window clipping in hardware:

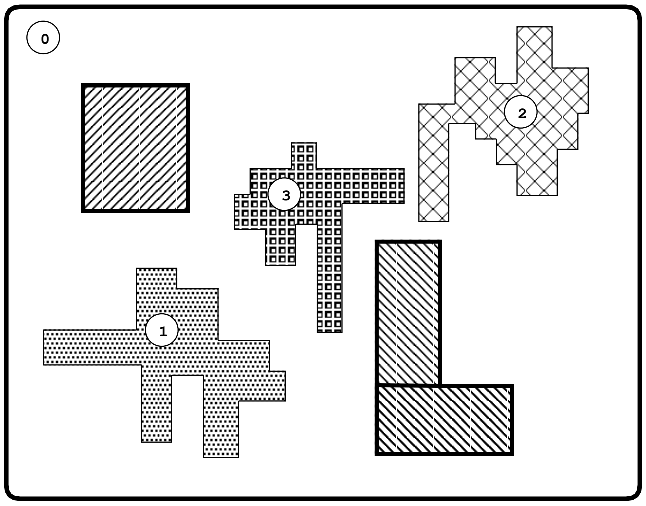

Figure 7: Clip rectangles define the clips for the two windows with simple drawable regions. One clip rectangle is needed for the rectangular window; two clip rectangles are needed for the L-shaped window. Three clip IDs are assigned to the three windows with irregular clips. Notice that clip ID zero is used as a "slop" clip ID for the screen region not assigned to the clip ID windows (which includes the windows clipped using clip rectangles).

Clip rectangles define the window's clip as a limited number of rectangles (1 to 4 available clip rectangles is typical). Windows with simple clips, representable with the available number of available clip rectangles, can be clipped this way.

Clip ID planes encode per-pixel state about the window to which a given frame buffer pixel belongs to. The hardware rendering state maintains a current clip ID value that can be compared with the clip value in the clip ID planes for every generated OpenGL fragment. When clip ID clipping is enabled, the fragment is rendered only if the current clip ID matches the pixel's clip value. Typically, 2 to 4 bits of clip ID planes are supported, providing 3 to 15 clip ID values.[5]

Clip ID planes work for arbitrary clips,[6] while clip rectangles work only for simple clips. Clip ID planes are generally slower than clip rectangles since every fragment clipped using clip ID planes must be tested against the pixel's clip ID that must be retrieved from frame buffer memory. Because the case of an unobscured window is the most important performance case *and* arbitrary clips must be supported, a hybrid approach combining

[5] Four clip ID planes would provide 16 possible clip values, but one clip value is reserved for the screen region not assigned to any particular clip ID assigned window, leaving only 15 clip values assignable.

[6] With the X Window System's Shape extension, arbitrarily shaped windows are easy to create, but even in a window system that only supports rectangualar windows, complex overlappings can still define arbitrary clips so the case of arbitrary clips must be addressed.

both clip rectangles and clip ID planes is common. Figure 7 shows clip rectangles and clip ID planes used simultaneously.

The clip ID and clip rectangle hardware state must be maintained and updated as necessary by the X server when window clips change. Note that there are a limited number of clip ID values so the X server must be ready to virtualize clip ID usage if too many windows have complex clips that require clip IDs.

Hardware Cursors

Window systems use cursors to track mouse motion. A *hardware cursor* means the hardware provides special support for merging the current cursor image at the current cursor location into the displayed frame buffer image. If a hardware cursor is not supported, the X server must draw the cursor image into the frame buffer. When rendering near or under a software rendered cursor, the cursor must be undrawn and redrawn between rendering commands. Not only does this result in a flickering cursor, but it means programs rendering to the hardware must know the cursor location to correctly draw and undraw the cursor. This greatly complicates 3D rendering, so hardware cursors are extremely common for accelerated OpenGL graphics hardware.

Graphics Subsystem Bottlenecks

A *bottleneck* occurs when one task in a system limits the overall system performance to the extent that other tasks in the system cannot reach their performance potential. There is inevitably some task or tasks that limit system performance, but the goal is to balance the limitations to avoid over-engineering some tasks while other tasks become bottlenecks.

The motivation for adding hardware to accelerate OpenGL is to achieve better overall graphics performance. But if hardware is added in an unbalanced fashion, expensive hardware resources will go underutilized due to bottlenecks in the system and overall performance will suffer.

Host vs. Transform vs. Fill Limits

Looking at the general `GTXRD` graphics pipeline, there are three broad types of graphics performance bottlenecks:

Fill limits occur in the rasterization stage. Fill limited programs are limited by the rate that pixels in the frame buffer can be updated. Visual simulation applications like flight simulators that draw large, textured polygons are often fill limited. The hardware fill limit is often expressed in millions of pixels per second, and the limit depends on what rasterization modes (like depth buffering) are enabled.

Transform limits occur in the transformation stage. Transform limited programs are limited by the rate that vertices can be lighted, transformed, and clipped. A molecule visualization program that renders thousands of atoms as finely tessellated spheres is likely to be transform limited. The hardware transform limit is often expressed in thousands of polygons per second, and the limit depends on what transformation modes (like the number of lights) are enabled.

Host limits occur in the generation and traversal stages. Host limits reflect the maximum rate at which the host computer can generate graphics commands so they can be processed by the graphics subsystem. Some tasks like the finite element analysis of a car crash cannot be done in real-time and are therefore clearly host limited, but even programs designed to be interactive may starve a fast graphics pipe if the program has too much overhead.

Notice that what limits a program's performance depends in large part on the application. Some applications may demand a machine with excellent fill performance; another application may demand a machine with excellent transform performance. A system architect must make compromises based on the machine's intended purpose.

A good technique for determining if a program is fill limited is to resize the window down to a very small size. If the program speeds up, the program is very likely fill limited. Likewise, if resizing the window to the full screen size has no effect on the program's graphics performance, the program is likely to be host or transform limited. Host based profiling tools can help you determine if a program is host limited. A specialized debugger for OpenGL called `ogldebug` is available to help characterize what operations an OpenGL program stresses.

Other limits that do not cleanly fit into the above categories are also possible. If programs change OpenGL state frequently enough, these mode changes become a bottleneck. Downloading texture maps into the hardware is another potential bottleneck.

Understanding the bottlenecks for a given application on a given system is important for tuning the application. A program that is fill limited may be able to utilize better lighting and finer tessellation without affecting application performance. Likewise, a transform limited program may achieve better performance by limiting the number of polygons transformed by using coarser tessellations.

OpenGL Command Bandwidth

Another performance bottleneck is the rate at which commands are transferred from the host to the OpenGL implementation. The easiest way to improve OpenGL command bandwidth is to use direct rendering if the application is running on the local graphics workstations.

One way to minimize the OpenGL command bandwidth is to use OpenGL display lists. When rendering indirectly, display lists are downloaded into the X server so that when executed, only the command to execute the display list must be transferred to the X server. This can greatly reduce the network overhead of indirect rendering. Display lists may also be downloaded into the graphics hardware so they can be executed with no host overhead to drive the display list.

It is also possible for a fast computer and fast graphics subsystem to be starved by the interface between them. Be careful that the host's peripheral bus used to connect the graphics subsystem has sufficient bandwidth to drive the graphics at your required performance.

Beware of Benchmarks

Beware of graphics benchmarks. The performance that matters is the performance of your applications. Graphics benchmarks are often difficult to interpret because not enough parameters are specified. For example, a vendor may claim "10 million 3D lines per second." While this number is high, it does not say if the lines are depth buffered, if the lines are connected or unconnected,[7] if the lines are shaded or textured, etc. Also, if your application is rendering polygons, line rendering speed is probably irrelevant.

Graphics benchmarks often rely on synthetic workloads. Real graphics programs do not simply render millions of 3D lines. Instead, real programs switch between various primitives and use various modes, and this switching overhead is rarely measured by benchmarks. Real programs also spend time generating the data to render, a task not done in benchmarks. Benchmarks are useful for determining the "not to be exceeded" rendering performance for a particular rendering task, but do not expect real applications to achieve benchmark results.

Conclusions

Rendering OpenGL graphics is a highly specialized, performance critical task that is suitable for hardware acceleration. The general graphics pipeline provides a good way to classify graphics hardware architectures. OpenGL is typically accelerated by performing some or all of the rasterization and transformation pipelined stages with specialized graphics hardware. Good window system support justifies hardware features other than those required for fast rendering. An evaluation of a graphics hardware subsystem for OpenGL should focus on whether the OpenGL implementation will have balanced performance for the applications intended for the hardware. Finally, the faster the graphics hardware, the more important support for OpenGL direct rendering becomes.

References

[1] Kurt Akeley, "The Silicon Graphics 4D/240GTX Superworkstation," *IEEE Computer Graphics and Applications*, July 1989.

[2] Kurt Akeley, "RealityEngine Graphics," *Proceedings of SIGGRAPH '93*, July 1993.

[3] Michael Deering, "Leo: A System for Cost Effective 3D Shaded Graphics," *Proceedings of SIGGRAPH '93*, July 1993.

[4] Richard Ferraro, *Programmer's Guide to the EGA and VGA Cards, 2nd Edition*, Addison-Wesley, 1990.

[5] Chandlee Harrel, Farhad Fouladi, "Graphics Rendering Architecture for a High Performance Desktop Workstation," *Proceedings of SIGGRAPH '93*, July 1993.

[6] Mark Kilgard, "Programming X Overlay Windows," *The X Journal*, SIGS Publications, July 1993.

[7] Mark Kilgard, David Blythe, Deanna Hohn, "System Support for OpenGL Direct Rendering," *Proceedings of Graphics Interface '95*, 1995.

[8] John Rohlf, James Helman, "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics," *Proceedings of SIGGRAPH '92*, July 1994.

[9] Mark Segal, Kurt Akeley, *The OpenGL Graphics System: A Specification*, Version 1.0, Silicon Graphics, June 30, 1992.

[10] Silicon Graphics, "Indy Graphics," *Indy^{TM} Technical Report*, Version 1.0, 1993.

[11] Paul Strauss, Rikk Carey, "An Object-Oriented 3D Graphics Toolkit," *Proceedings of SIGGRAPH '92*, July 1992.

[7] Rendering N unconnected lines requires $2 \times N$ vertices; rendering connected lines means each line starts where the last one ended so only $N + 1$ vertices are required. Rendering connected lines requires half the per-vertex bandwidth of unconnected lines.