SIGGRAPH 2004
Course # 16

Performance OpenGL: Platform Independent Techniques

or

"A bunch of good habits that will help the performance of any OpenGL program"

Tom True
ttrue@nvidia.com

Brad Grantham
grantham@sgi.com

Bob Kuehne
rpk@blue-newt.com

Dave Shreiner
shreiner@sgi.com

Course Description

The OpenGL Application Programming Interface (API) is the most widely supported, cross-platform computer-graphics interface available to programmers today. Such broad support of OpenGL across different graphics hardware presents challenges in maximizing performance while still maintaining portability. This course focuses on teaching tools and techniques for the analysis and development of high-performance interactive graphics-applications.

The course begins with an introduction to the stages of the OpenGL rendering pipeline, and the performance-limiting *bottlenecks* [1] that can affect each stage of the pipeline. Simple, yet effective, techniques for determining which stage of the pipeline is limiting are presented.

The course continues with a detailed analysis of each stage's operations, and the options that a programmer has control over. The interaction of those options with respect to performance is discussed, with an eye towards highlighting commonalities that aid performance across different platforms is presented along with empirical results to back up our observations.

A brief departure into topics relating to object-oriented programming paradigms and how one's coding style can effect OpenGL's operation. Some suggestions for managing data in OpenGL with respect to encapsulation are presented.

Continuing, the course introduces the concept of OpenGL's *validation* phase, and how this hidden operation can rob perfectly legitimate, well-written OpenGL applications of their performance. This discussion culminates with a discussion of *state sorting*.

Finally, simple suggestions that are likely to help the performance of OpenGL applications are presented. These suggestions should be added into the programmer's tool box, providing simple optimizations that do not violate Professor Knuth's old adage of "premature optimization is the root of all evil."

Course Prerequisites

This course assumes that the attendee has a basic understanding of computer graphics techniques (e.g., depth buffering, texture mapping, etc.), and of the OpenGL programming interface.

Additionally, a brief understanding of object-oriented programming would aid in the understanding of the encapsulation discussion, but is by no means required.

[1] *Italicized* terms are defined in the glossary section

Introduction

OpenGL is the most widely accepted cross-platform API for rendering computer-generated images interactively. Its simplicity as a programming library allows the novice OpenGL application developer to quickly develop applications capable of rendering complex images containing lighting effects, texture mapping, atmospheric effects, and anti-aliasing, among other techniques. However, the low-level nature of the OpenGL API also makes it well suited for the experienced developer to author interactive real-time applications. As with any programming language, there's more than one way to code a solution, however, not all solutions exhibit the same performance. The same is true of OpenGL applications; perfectly valid OpenGL programs that render the same image may have drastically different performance characteristics, based only on how the OpenGL calls were made in the application.

These notes will illustrate some of the conditions that can hamper OpenGL performance in a platform-independent fashion and present strategies to determine which methods for rendering pixels and geometry are optimal for particular platforms without reverting to platform-specific extensions.

Many of the techniques discussed in this paper can be explored in the OpenGL Sample Implementation (SI) [5]. The SI is a software-only rendering implementation of the OpenGL interface. This source base was provided to OpenGL licensee's before its open-sourcing, and as such, may provide some insight into the construction of many of the hardware drivers available today.

To determine what are the *fast paths* for a particular OpenGL implementation, there is truly only one real way to determine credible results (if you share the same mistrust of marketing information that we do): develop a short program that tests the features that you intend to use. Although the techniques discussed in this paper are generic, every platform will have its own "sweet spot(s)". As such, OpenGL provides information that can be queried at runtime, namely the `GL_VERSION`, and `GL_RENDERER` strings, that can help you make determinations as to the combinations of techniques that will provide the best performance for your particular platform.

Errors and OpenGL

First and foremost, verify your application doesn't have any OpenGL errors

Errors can have a drastic impact on the performance of an OpenGL application. OpenGL was designed to maximize interactive performance, and therefore does not actively report errors that an application might make when calling into the OpenGL interface. As OpenGL does not return errors after each function call, it falls to the application developer to check the OpenGL error state to determine if an error has occurred. It seems that many OpenGL developers don't bother to check for errors; this is understandable, however, as most situations that can cause OpenGL to emit an error will cause a visual difference in the image that is generated. Nonetheless, verifying the program has no errors is an important first step in tuning its performance.

The OpenGL specification mandates that no input into an OpenGL call should cause a catastrophic error (e.g., a "core" dump); calls made with invalid data are merely ignored (i.e., no alteration to the framebuffer or the OpenGL state is made). After the internal OpenGL error state is set to represent the situation that occurred, the OpenGL call immediately returns. The immediate return is the important point when analyzing OpenGL performance.

The effect that an error in calling an OpenGL function has on performance is that it artificially makes the application seem faster than it really is. Since the call that experienced the error never completes, the "work" that call was intended to do is never completed. For instance, `glTexImage2D()` is particularly susceptible to errors (it's easy to get a token incorrect, or mix the *type* and*format* parameters, when first writing a program) and does considerable work before its completion. The `glTexImage*D()` commands first transfer the texture data from host memory into the OpenGL pixel pipeline, allocate texture memory, convert the host-side data into the internal format requested, and finally, stores the texture in the allocated texture memory. If an error occurs, none of the previously mentioned work is completed, and the call seems to complete quite rapidly. If an application loads one texture per frame, and the `glTexImage2D()` call has an error each frame, the application will seem to execute considerably faster than it would without errors.

It is important to note that once an error has occurred, the OpenGL error state will not be updated until the state has been cleared (by checking for an error).

Just as there's little reason to optimize an application that crashes, verifying the performance of your OpenGL application while it has errors may cause you to optimize in the wrong sections of the application, or mislead you completely. Check for errors early and often in the development of the application; once you've determined that there are no more errors, move onto optimizing the program.

How to determine if you have any errors

Checking for OpenGL errors is simple. A call to `glGetError()` will return one of OpenGL's error codes, or `GL_NO_ERROR`. As an implementation may track more than one error (this in not in contradiction to the above statement about OpenGL only recording the first error; if the implementation is distributed, the specification permits each "part" of the implementation to maintain error state. Calls to `glGetError()` return errors in a random manner from each part in such a case).

A simple C-preprocessor macro can be very helpful in detecting errors for an individual call, such as illustrated in Example 1

Example 1 macro for checking an OpenGL call for an error

```
#define CHECK_OPENGL_ERROR( cmd ) \
  cmd; \
  { GLenum  error; \
    while ( (error = glGetError()) != GL_NO_ERROR) { \
      printf( "[%s:%d] '%s' failed with error %s\n", __FILE__, \
              __LINE__, #cmd, gluErrorString(error) ); \
  }
```

(This assumes an ANSI compliant C-preprocessor. Otherwise, replace the `#cmd` with `"cmd"` [including the quotation marks])

This macro can be placed almost anywhere in an application after the creation of an OpenGL context, with the exception of between a `glBegin()`/`glEnd()` sequence. However, calling `glGetError()` inside a `glBegin()`/`glEnd()` sequence, will itself generate an error. This presents a problem for error checking: if a `glBegin()` is not matched by an `glEnd()`, calling `glGetError()` to get the error state causes an error itself.

Available at the web site listed in the last section of these notes is a Perl script that will rewrite an OpenGL header file for automating OpenGL error checking. The script handles almost all OpenGL types of errors, and reports useful information about which OpenGL calls produced errors and where in the application's source code the error occurred.

Errors with Feedback and Selection Mode

Feedback mode (enabled by calling `glRenderMode(GL_FEEDBACK)`, and selection mode (enabled by calling `glRenderMode(GL_SELECT)` utilize a slightly different error reporting method than most OpenGL functions. For both modes, a user-allocated memory buffer is required. The most common error for either mode is that this buffer is too small and a buffer overflow condition occurs. In such a case, no error is reported when calling `glGetError()`, rather, the return value from the next `glRenderMode()` is a negative number. In normal operation, `glRenderMode()` returns the number of "slots" in the buffer that were filled.

Errors relating to vertex and fragment programs

With OpenGL version 1.4, two new extensions were introduced, `GL_ARB_vertex_program` and `GL_ARB_fragment_program`. These extensions are not part of the core OpenGL language, however, it's likely that modern graphics accelerators will support these extensions.

Vertex and fragment programs, as described by these extensions, can be thought of as compiled assembly language programs. The specific assembly language instructions are detailed in the extension specifications available at `http://oss.sgi.com/projects/ogl-sample/registry/index.html`. The compilation of the program occurs within your application when the function `glProgramStringARB()` is executed; no external compiler is required. This compilation phase could fail, and at that time, OpenGL will set the current error to `GL_INVALID_OPERATION`. Additional information about the error can be obtained by calling `glGetIntegerv()` with an argument of `GL_PROGRAM_ERROR_POSITION_ARB` that will return the byte offset from the beginning of the program where the error occurred. Additionally, `glGetString()` can be called with `GL_PROGRAM_ERROR_STRING_ARB` that will return an implementation-dependent error description. This string can be overwritten for each error, and may not be cleared if the error state is modified by other OpenGL operations, so both error state values should be checked simultaneously if an error occurred.

Compilation of a program is only half of the setup to use vertex and fragment programs with OpenGL. In order to "load" the program into OpenGL, the program must be bound with the `glBindProgramARB()` call. If the specified program id is not valid, this call will generate a

GL_INVALID_OPERATION error. If glBindProgramARB() fails, but the error condition is not checked, then the next glBegin() (or any call that might utilize glBegin(), like glDrawArrays()), and glRasterPos() will generate a GL_INVALID_OPERATION error. (Some current OpenGL implementations silently allow rendering with invalid vertex and fragment programs, so it's important to verify the success of glProgramStringARB() and glBindProgramARB().)

Various other errors associated with setting program parameters (using either the glProgramLocalParameter*ARB(), or glProgramEnvParameter*ARB() calls) can occur, as well as GL_OUT_OF_MEMORY errors if a program is larger than can be executed by the underlying hardware.

Application Performance Bottlenecks

Determine which part of the OpenGL pipeline is the performance limiting factor

Regardless of how well written an application is, one section will always execute more slowly than any of the rest of the program. This slowest section may be the part of the program parsing an input file, computing a complex mathematical formula, or rendering graphics. The slowest part of an application is often called the "bottleneck." Every application has a bottleneck. The question regarding bottlenecks is always whether they impact performance enough that they need to be tuned and their performance improved.

Since applications always have bottlenecks, tuning the section that is the slowest currently will generally cause the bottleneck to migrate to a different part of the application. This is to be expected; what is important is that after addressing the performance of the chosen bottleneck, is the application meeting the performance metrics that is it required to meet?

Overview of OpenGL Rendering Pipeline

The OpenGL rendering pipeline is the process that receives geometric and imaging primitives from the application and rasterizes them to the framebuffer. The pipeline can basically be split into two sections: *transformation phase* and *rasterization phase*.

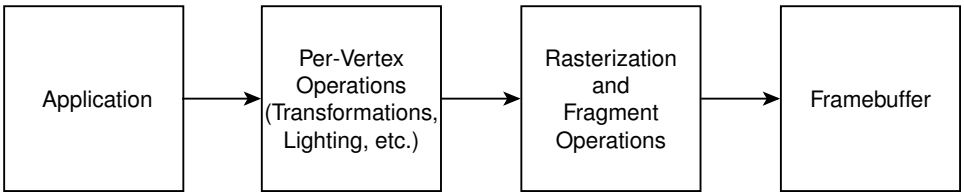

Figure 1: Block diagram of OpenGL pipeline phases

The transformation phase is responsible for vertex transformations, lighting, and other operations, and produces OpenGL *fragments* that are passed into the rasterization phase.

Rasterization is responsible for coloring the appropriate pixels in the framebuffer, which is more generally known as a process named *shading*. This phase also includes operations like depth testing, texture mapping, and alpha blending, among others.

The same rendering pipeline structure is present even with the latest programmability additions to OpenGL [2]. Some clarification may be useful. An OpenGL implementation that doesn't have programmability available in its feature set is said to have a *fixed-function pipeline* where the order of operations is strictly ordered. For example, for the transformation phase, vertices are:

1. transformed by the model-view matrix
2. if enabled, a color derived from the lighting state is computed and replaces the current vertex color
3. if enabled, texture coordinates are generated form the texture-coordinate generation state
4. transformed by the projection matrix
5. are divided by the w coordinate
6. clipped to the view volume, generating new vertices if necessary

[2] At least for the moment; it's foreseeable that in the future, rasterization operations, like texel lookup, might be available to provide data in a vertex program

7. passed to the rasterization phase of the pipeline

But in the case of vertex programs all the above per-vertex operations are executed in the vertex program and are no longer strictly ordered as in the fixed-function pipeline.

The rasterization phase undergoes a similar, though not as comprehensive change. Currently, fragment programs replace texel lookups and their application to the pixel (i.e., texture environment). One can also modify the depth value, and primary and secondary colors for the fragment, but this doesn't represent an exchange of features as in the vertex program case.

"Bottleneck" determination

Understanding how the OpenGL pipeline operates makes it much easier to understand in which section of your application or OpenGL code the performance problem, or "bottleneck" exists. Generally, bottleneck determination with respect to OpenGL consists of narrowing the problem down to one of three possibilities:

- the application can't draw all the pixels in the window in the allocated time. This situation is known as "fill limited."
- the application can't process all of the vertices that are required for rendering the geometric objects in the scene. This situation is known as "vertex (or transform) limited."
- the application can't send the rendering commands to OpenGL rapidly enough to keep OpenGL's renderer busy. This final situation is known as "application limited."

The next sections discuss simple techniques to determine which of the above conditions exist in the application. If the application doesn't meet the performance criteria (which should be a metric such as frames per second, objects per second, or some other quantitative value), then one of the above situations will always exist.

From a practical standpoint, the application bottleneck (principally due to bus speeds) is becoming the most predominant condition as the speed of graphics accelerators improves. As such, the placement of data and its formatting will become much more relevant, as we'll soon see.

Techniques for determining pixel-fill limitations

Fill-limited applications are limited by being unable to color all the pixels in the viewport in the allotted time. This situation results from too much work being required by some or all pixels in the generated image, or simply rasterizing too many pixels.

Of the three performance situations, pixel-fill limitations are perhaps the simplest to identify. The fundamental problem is the viewport has too many pixels being filled, so if the application's performance increases when the viewport is decreased in size, the application is fill limited.

Two solutions are immediately evident: reduce the size of the viewport for the applications execution, or do less per-pixel processing. In some cases, such as the design requirements for the application, reducing the size of the viewport may not be an option. A full-screen window has to be the size of the full screen. It may be acceptable to reduce the monitor's resolution but that still may be outside the latitudes of the program's requirement. The more common solution is to reduce the amount of work per-pixel. Techniques for reducing per-pixel work are discussed in the optimization section.

Techniques for determining transformation limitations

If reducing the size of the viewport doesn't increase the frame-rate of the application then it is unlikely that the application is fill- limited. In such a case, a simple test can be conducted to determine if the bottleneck is in the transformation section of the rendering pipeline or caused by other non-OpenGL portions of the application.

To determine if the application is vertex limited convert all calls from `glVertex*()` to `glNormal*()`. This can be accomplished quite simply by utilizing the C preprocessor once again, making substitutions as demonstrated in Example 2:

Example 2 macros for replacing `glVertex*()` calls with `glNormal*()` calls for determining application data-transfer speed

```
#define glVertex2d(x,y)       glNormal3d(x,y,0)
#define glVertex2f(x,y)       glNormal3f(x,y,0)
#define glVertex2i(x,y)       glNormal3i(x,y,0)
#define glVertex2s(x,y)       glNormal3s(x,y,0)
#define glVertex3d(x,y,z)     glNormal3d(x,y,z)
#define glVertex3f(x,y,z)     glNormal3f(x,y,z)
#define glVertex3i(x,y,z)     glNormal3i(x,y,z)
#define glVertex3s(x,y,z)     glNormal3s(x,y,z)
#define glVertex4d(x,y,z,w)   glNormal3d(x,y,z)
#define glVertex4f(x,y,z,w)   glNormal3f(x,y,z)
#define glVertex4i(x,y,z,w)   glNormal3i(x,y,z)
#define glVertex4s(x,y,z,w)   glNormal3s(x,y,z)

#define glVertex2dv(v)        glNormal3d(v[0],v[1],0)
#define glVertex2fv(v)        glNormal3f(v[0],v[1],0)
#define glVertex2iv(v)        glNormal3i(v[0],v[1],0)
#define glVertex2sv(v)        glNormal3s(v[0],v[1],0)
#define glVertex3dv(v)        glNormal3dv(v)
#define glVertex3fv(v)        glNormal3fv(v)
#define glVertex3iv(v)        glNormal3iv(v)
#define glVertex3sv(v)        glNormal3sv(v)
#define glVertex4dv(v)        glNormal3dv(v)
#define glVertex4fv(v)        glNormal3fv(v)
#define glVertex4iv(v)        glNormal3iv(v)
#define glVertex4sv(v)        glNormal3sv(v)
```

The same technique can be used to replace the calls to `glRasterPos*()`, however, `glRasterPos*()` only invokes a single vertex transformation. Most of the real is work done when the subsequent `glBitmap()` or `glDrawPixels()` calls are made, which have little impact on transformation bottlenecks.

The reason that this technique is useful is that it transfers the almost same quantity of data from the application to the OpenGL pipeline, however, no geometry is ever rendered (`glNormal*()` sets the current normal in the OpenGL state, so its work is limited to a few memory store operations). Additionally, `glNormal*()` calls are not subject to "hidden" state changes, as you might find with `glColor*()` while `GL_COLOR_MATERIAL` mode is enabled.

However, the situation is more difficult if an application uses vertex arrays. The calling of `glVertex*()` in the case of vertex arrays isn't explicit so the command substitution trick doesn't work. While it is possible to disable the vertex data (using `glDisableClientState(GL_VERTEX_ARRAY)()`, this isn't terribly useful, since the vertex data is not being transferred to the pipe, which is what we are ultimately trying to measure. If another array is not being currently utilized, like `GL_TEXTURE_COORD_ARRAY`, one could specify the vertex data as three-dimensional texture coordinates. The same amount of data would be transferred to the graphics hardware, but nothing would be rendered. One other caveat, however, is that the OpenGL specification [6] permits multiple valid implementations of vertex arrays; that is, in some cases, vertex arrays are merely wrappers around the immediate mode calls, such as `glVertex*()`. In other implementations, it's quite possible that the data is

processed on the graphics hardware without the execution of immediate mode calls. In either case, it's difficult to know exactly what the particular OpenGL implementation will do in the absence of vertex data. Empirical tests are required for certain determination on your target platforms.

To generate a useful performance metric, measure and record the time of rendering a frame as normal. Then, after making the above replacements, time the same rendering operation. If the times are significantly different, the application is transform limited (assuming it's not fill limited). If the times are not appreciably different, and the fill limit test doesn't decrease the rendering time, then the data structures and formats chosen by the application are more likely the bottleneck.

Knowing when it's your fault

If the application is neither fill nor transform limited, then the problem lies in the choice of data structures and programming techniques employed in the application proper. As this situation is clearly outside of what can be addressed in the context of OpenGL, the best recommendation is to review the choices made for data structures, particularly for those that store data that is to be passed to OpenGL. General advice in this realm revolves around attempting to minimize pointer-dereferences and indirection, ensuring that data structures pack-well into memory, and that computation is as limited as feasible during OpenGL rendering. Begin investigation into application-limited code by using performance measurement tools (such as the `pixie` and `prof` available in versions of the Unix™operating system) to help locate and tune these bottlenecks.

Performance Optimizations for each stage of the OpenGL pipeline

Techniques for reducing per-pixel operations

We will start with an analysis of the rasterization section of the OpenGL pipeline. Rasterization has many stages that "fragments" (OpenGL's concept of a pixel, but with more information than just position and color) must traverse before being written into the framebuffer. By first understanding the application's visual quality requirements and interactive performance requirements, you then have a baseline metric to use as a control to optimize what combinations of fragment processing stages are employed in the application.

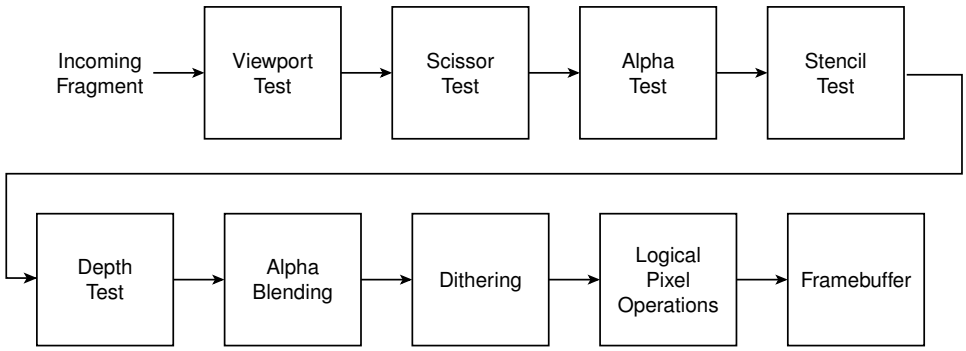

Figure 2: Stages of fragment processing pipeline

The operations in the rasterization pipeline can be classified into two categories: color-application (shading) processes and fragment testing.

- color-application processes include texture mapping, alpha blending, per-pixel fogging, stencil buffer updates, dithering, logical operations and accumulation buffer techniques. fragment programs fall into this category as well.
- fragment testing operations include depth buffering, stencil tests, alpha-value tests, and viewport and scissor box tests.

Reducing the visual quality of the rendered image is one method of reducing the fill limitation. This can be accomplished in a number of ways, not all of which may be suitable for the requirements of the application.

Be cognizant of the costs of per-fragment operations

Perhaps the simplest way to handle fill bottlenecks is to reduce the amount of work done per pixel. In some cases, this can be easily accomplished:

- don't blend transparent fragments (transparent where $\alpha = 0$). Eliminate their processing by enabling alpha fragment testing.
- disable depth-testing when feasible. Situations where this is useful include:
  - "clearing" the viewport using geometry (e.g., a "sky-earth" model where large gouraud shaded polygons are used to clearing the window. `glClear(...| GL_DEPTH_BUFFER_BIT)` must still be called to clear the depth buffer, but the polygons rendered need not be depth tested. This can be achieved by disabling the depth test.

    Although this is less important on recent OpenGL implementations that use a fast clear (sometimes called a "tag clear"), in which the clear operation is optimized, if your application will be rendering

a background polygon or shape anyway which covers the viewport, it is probably better to simply render the background object without depth testing to avoid the clear.

– depth-sorted polygons are rendered in order. This is what's classically called the *Painters' Algorithm*. Although polygon sorting along the line-of-sight, which is what's required for this technique, can be difficult, it can considerably reduce the amount of depth testing that's required. Additionally, other algorithms, like *binary space partition trees*, are well suited for use in depth sorting.

  However, a new hardware technique, generally referred to as *hierarchical z-buffering*, may cause the previous statement to be proven incorrect. Hierarchical z-buffering discards blocks of fragments if it determines that all of the fragments will be obscured what's already present in the framebuffer. In such a case, it's advantageous to render your geometry from front-to-back, which is basically the inverse of the Painters' algorithm.

The following results demonstrate the reduction in pixel-fill performance as more features are enabled. For the following graphs, every combination of enabling: 2D texture mapping, depth testing, alpha blending, dithering, stencil testing, and alpha testing, was conducted and a value recorded. The performance values were then sorted in a descending manner and the percentage difference between the two values were computed to form the second set of graphs. Enabling texture mapping causes, across most implementations, at least a 30% performance degradation from peak pixel-fill rates.

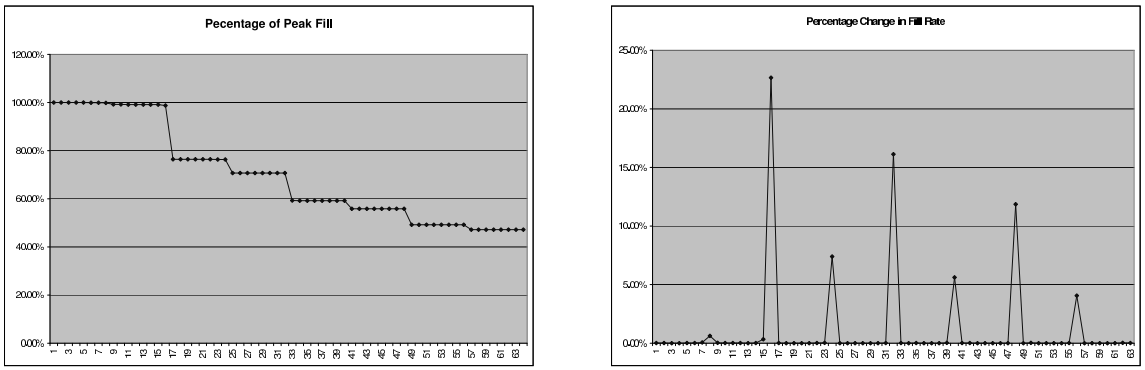

Figure 3: Fragment processing Results for Machine 1

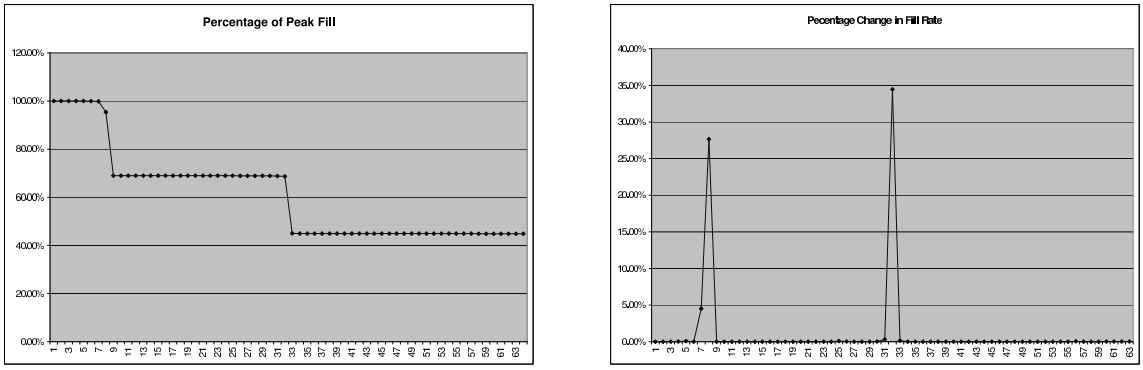

Figure 4: Fragment processing Results for Machine 2

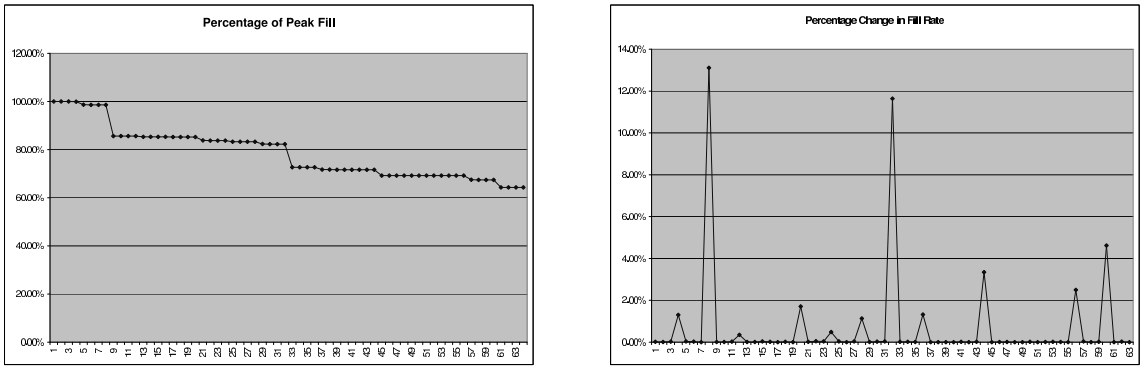

Figure 5: Fragment processing Results for Machine 3

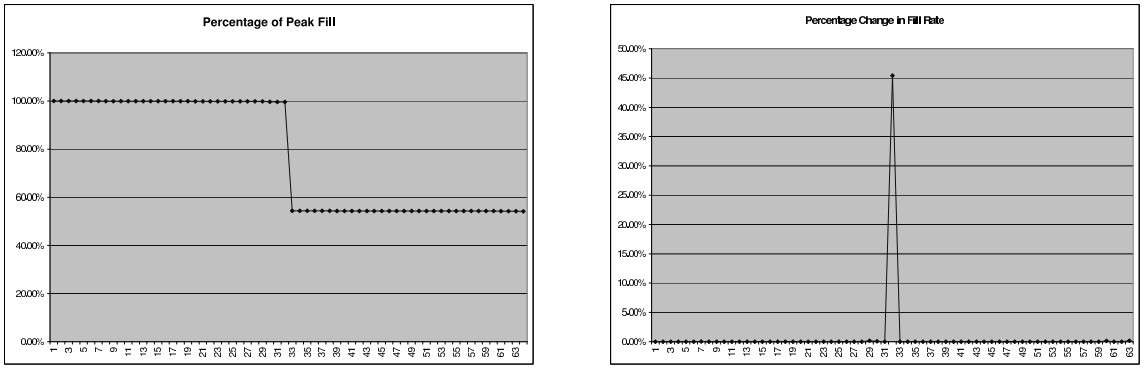

Figure 6: Fragment processing Results for Machine 4

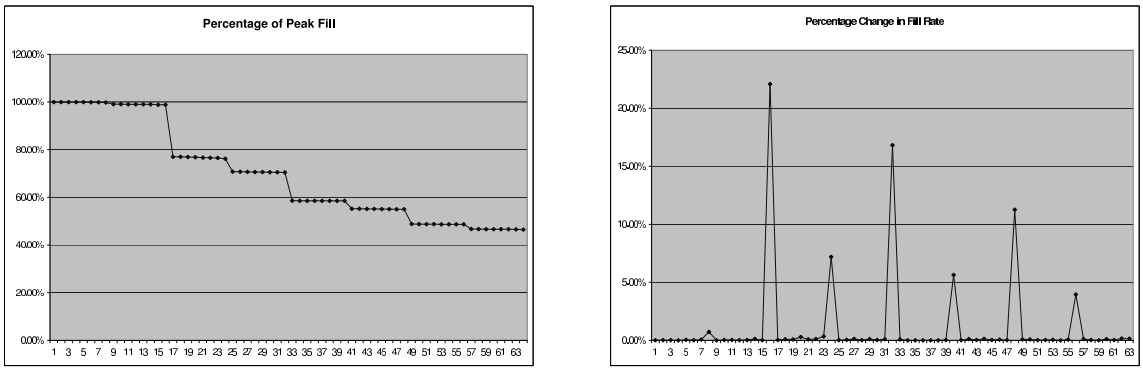

Figure 7: Fragment processing Results for Machine 5

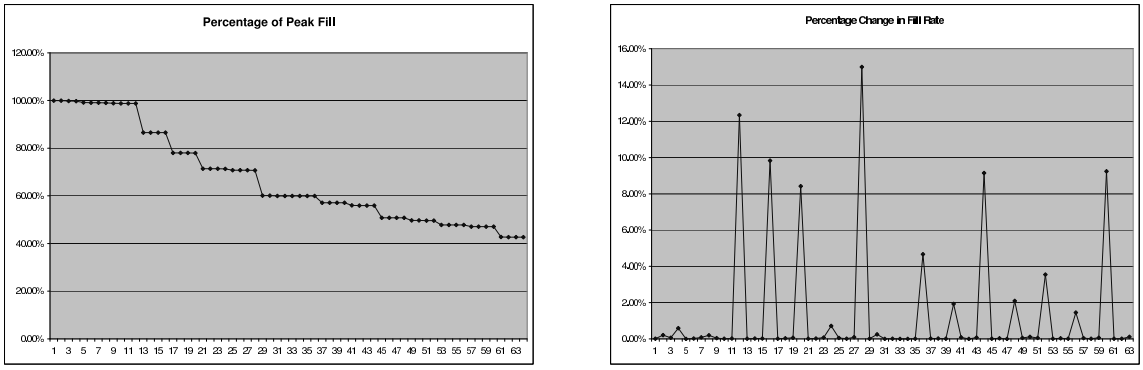

Figure 8: Fragment processing Results for Machine 6

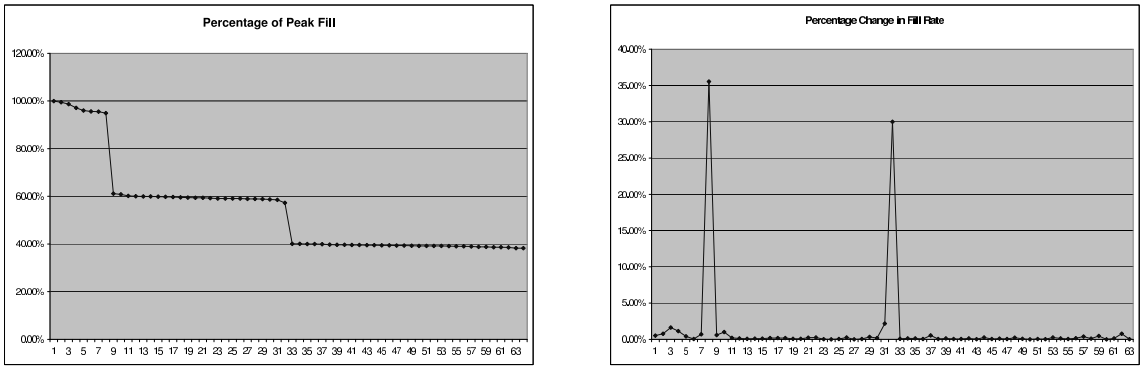

Figure 9: Fragment processing Results for Machine 7

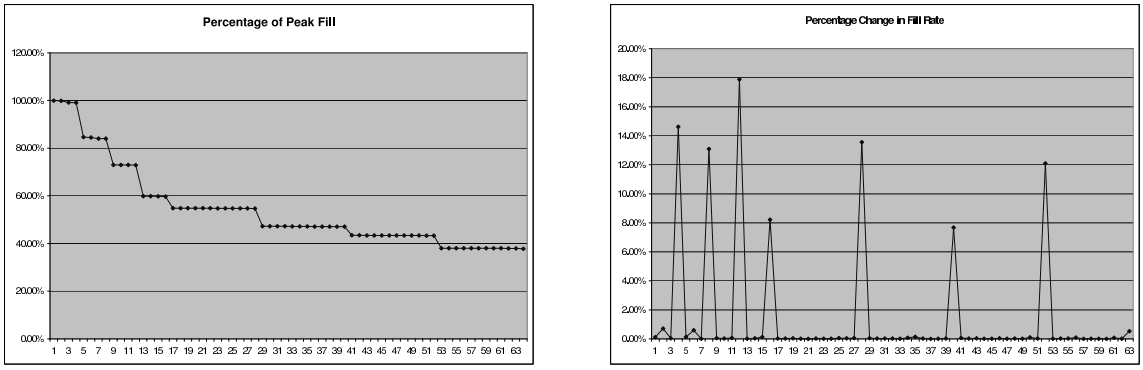

Figure 10: Fragment processing Results for Machine 8

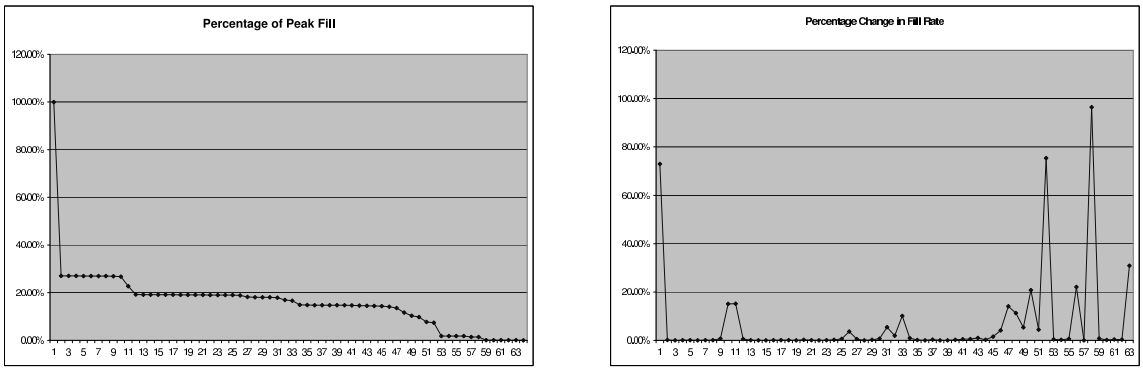

Figure 11: Fragment processing Results for Machine 9

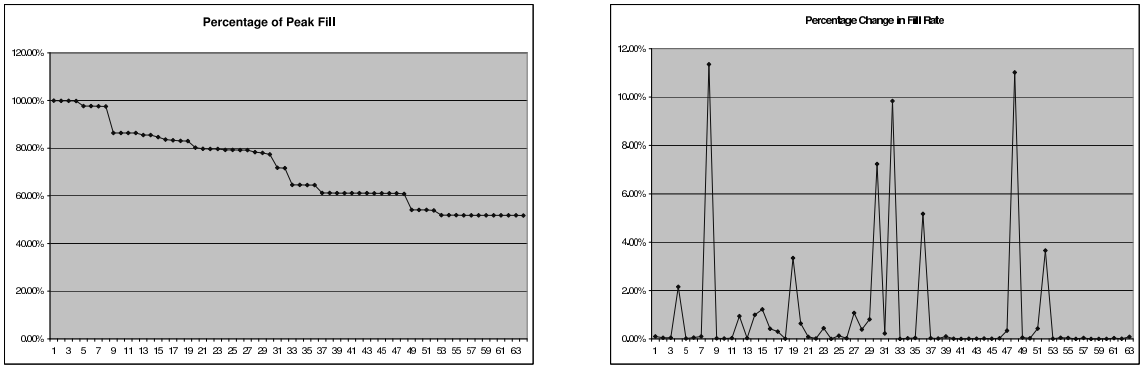

Figure 12: Fragment processing Results for Machine 10

Appendix A contains tables specifically listing the modes that correspond to each index (values on the X-axis). In many cases, with one mode enabled, all other modes are "free" (meaning their impact is negligible).

Reduce the number of bits of resolution per color component.

Reducing the number of colors that are available per component, decreases the number of bits that need to be filled. For example, reducing the framebuffer depth from 24 bits TrueColor to 15 bits TrueColor for a 1280x1024 sized window, reduces the number of bits that need to be filled by 37.5% (1.25 MBs). The down side to this approach is that you lose considerable color resolution, which may result in banding in the image. Dithering can help to reduce the banding but adds both some additional fragment processing and additional visual artifacts.

Reduce the number of pixels that need to be filled for geometric objects

Backface culling is a technique to determine which faces of a geometric object are facing away from the viewer, and under the appropriate conditions, do not need to rendered. This technique really trades fill rate for geometric transformation work. Specifically, the given polygon is transformed, and then the signed screen-space area is computed. If that value is negative, then the polygon is back-facing. By utilizing this technique, the amount of depth buffering required by the application is reduced. However, not every object is suited for backface removal. The best candidates are convex closed objects (e.g., spheres, cones, cubes, etc.).

Utilize a lesser quality texture mapping minification filter

OpenGL supports six modes for filtering texels for application to a pixel under minification. The mode that provides the best visual quality, `GL_LINEAR_MIPMAP_LINEAR`, requires the computation of an eight-way weighted average (for the 2D texture-mapping situation) to arrive at a single fragment color value. The most conservative minification mode is `GL_NEAREST`, which only requires a single texel lookup per fragment. The tradeoff, once again, is visual quality.

In comparing modern graphics hardware, some interesting conclusions can be drawn. The following data result from rendering the same scene using each of the minification filters available. The following set of graphs illustrate rates for 64×64 texel textures.

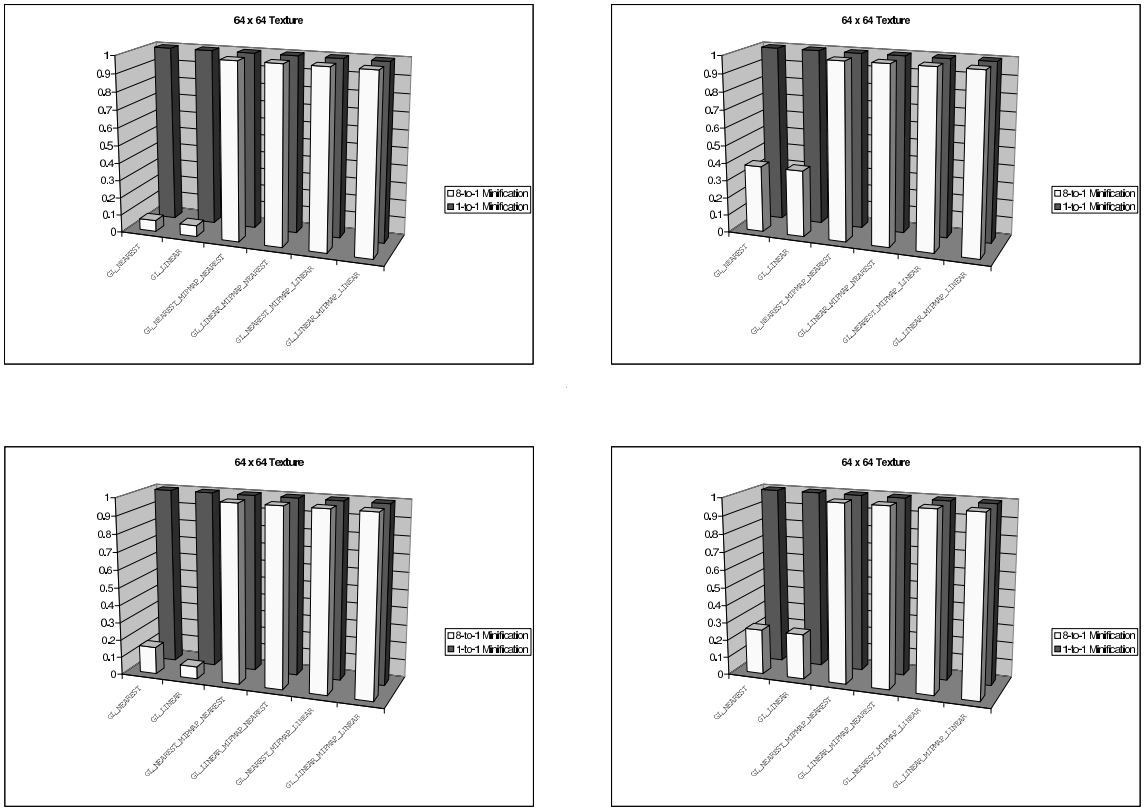

Figure 13: Results for 64×64 sized textures

Additionally, to test to determine if there are caching affects on performance, a 512×512 texture was utilized for the following results:

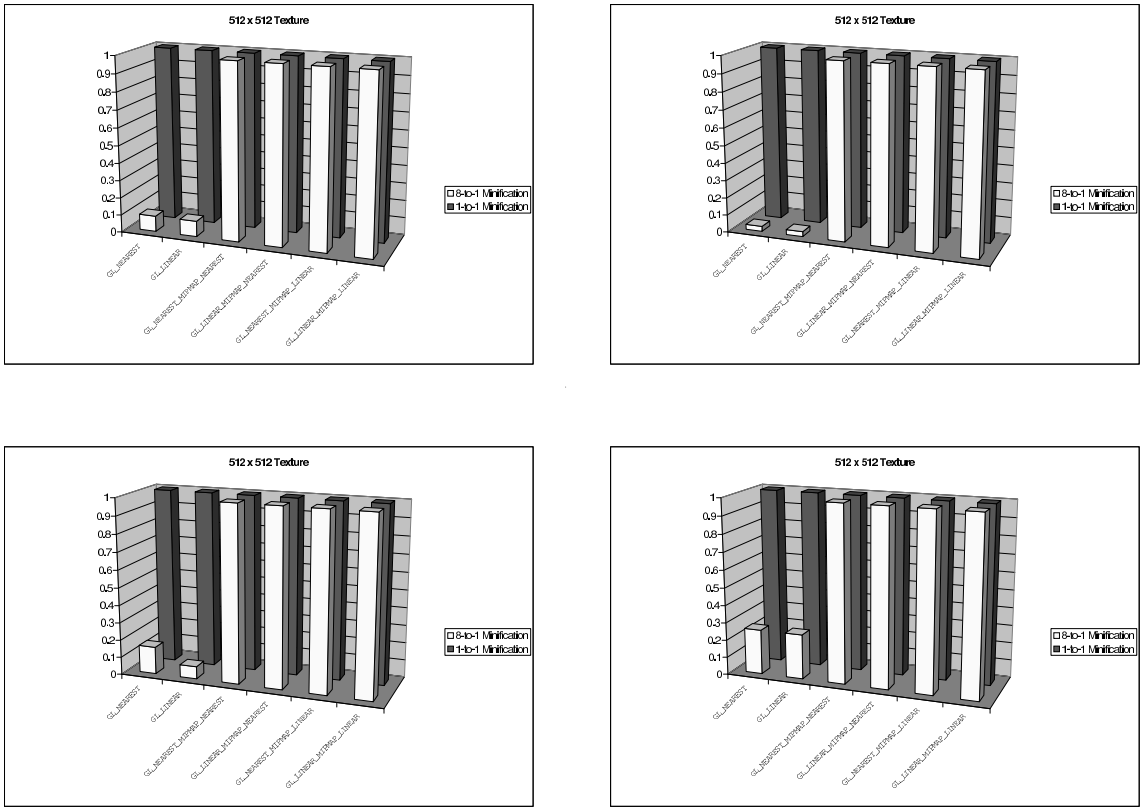

Figure 14: Results for 512×512 sized textures

We can see from the above graphs that pixels that require considerable minification (represented by the "8-to-1 Minification" in the graphs), the simplest filters `GL_NEAREST` and `GL_LINEAR`, perform considerably slower than the mipmap based filters. Interestingly, in previous years, the best advice to maximize texture-mapping speed was to use the least taxing minification filter: `GL_NEAREST`. It would now seem that that advice no longer holds as much relevance as it did previously.

However, when considerations are made for the screen-space size of the object, it may become useful to change the minification filter as the size of the object decreases. This is a level-of-detail technique that requires tracking the object's screen-space presence. Utilizing this technique may well help the rendering speed of the application, but may transfer some of the burden to the application data processing, which may in turn become the bottleneck.

Reduce the number of depth comparisons required for a pixel

Depth testing is an expensive operation. For each fragment, a read of the depth buffer is required, then a comparison, and if successful, then a write into the depth buffer, and finally, the color update to the framebuffer. In diabolical situations, the actual number of pixels that make it to the screen may be a minute fraction of those sent through the rasterization pipeline.

One method to determine the application's depth-buffer efficiency, is to utilize "hot-spot analysis." Hot-spot analysis can be used to determine which regions of the scene are causing an extreme amount of depth buffering to

occur, and then perhaps, a data management technique, like occlusion culling may be brought into play to reduce the pixel complexity of that region of the image.

To generate an image for hot-spot analysis, some modification of the application's rendering procedure is required. The fundamental technique is to clear the framebuffer to black, render all of the geometry in a uniform color (say, white) in the scene with depth buffering disabled, and blending enabled with the blending modes set with a call to
`glBlendFunc( GL_SRC_ALPHA, GL_ONE )`, with the alpha value set to suitably small number. In the resulting image, areas that appear more white, have higher incidences of depth testing, and indicate area where a culling algorithm on geometry may be useful.

Utilize per-vertex fog, as compared to per-pixel fog

As per-pixel fog adds an additional color interpolation operation per pixel, utilizing per-vertex fog, may reduce the per-fragment computation cost. This mode can be enabled by calling `glHint( GL_FOG_HINT, GL_FASTEST )`.

Techniques for reducing per-vertex operations

When the bottleneck of the application occurs in the transformation stage, too much work per-vertex (from some set of vertices) is required. Like the rasterization section of the OpenGL pipeline, the transform pipeline also contains several features that can be manipulated to control application performance.

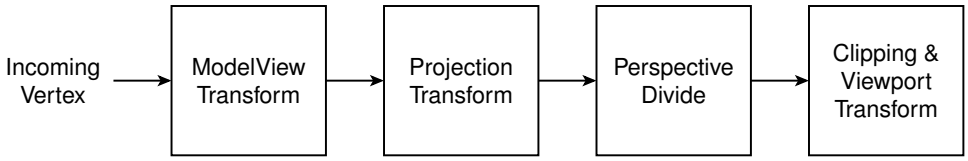

Figure 15: Stages of transformation pipeline

Be cognizant of the work done at a vertex

The work required to process a vertex can vary greatly particularly if features like lighting or texture-coordinate generation are enabled. Every vertex is processed by the model-view and projection matrices, clipped to the viewing frustum, and finally projected to the viewport by the viewport transformation. This is quite a bit of mathematical processing even with no additional features enabled. There are a few ways to minimize this mandatory processing that are described in the next sections.

One of the largest contributors to per-vertex work is lighting. The OpenGL lighting equation in its fullest form is computationally demanding. Each light that is enabled contributes to the per-vertex work. In many OpenGL hardware implementations, not all lights that are supported by OpenGL are implemented in hardware. Additionally, the type of light may require different amounts of work. OpenGL supports two general types of lights: *point* [or *local*] lights, and *infinite* [or *directional*] lights). The following graphs illustrate the performance drop-off that occurs when hardware lighting cannot accommodate all the lights requested. In some cases the penalty for falling to the software path is quite drastic — the graphics accelerator may either disable any hardware support for lighting, or it may be required to send data back to the host representing the lighting values that were computed in the hardware, and then combine those with the software evaluation of the remaining lights.

This sequence of results represent local-light illumination of one-pixel- area triangles. Empirical results indicate that the same rendering sequence using `GL_POINTS` is not optimized on almost all of the graphics platforms tested. In many cases, transforming three times as many vertices (as is the case in using triangles as compared to `GL_POINTS`), was faster than rendering single pixel `GL_POINTS`.

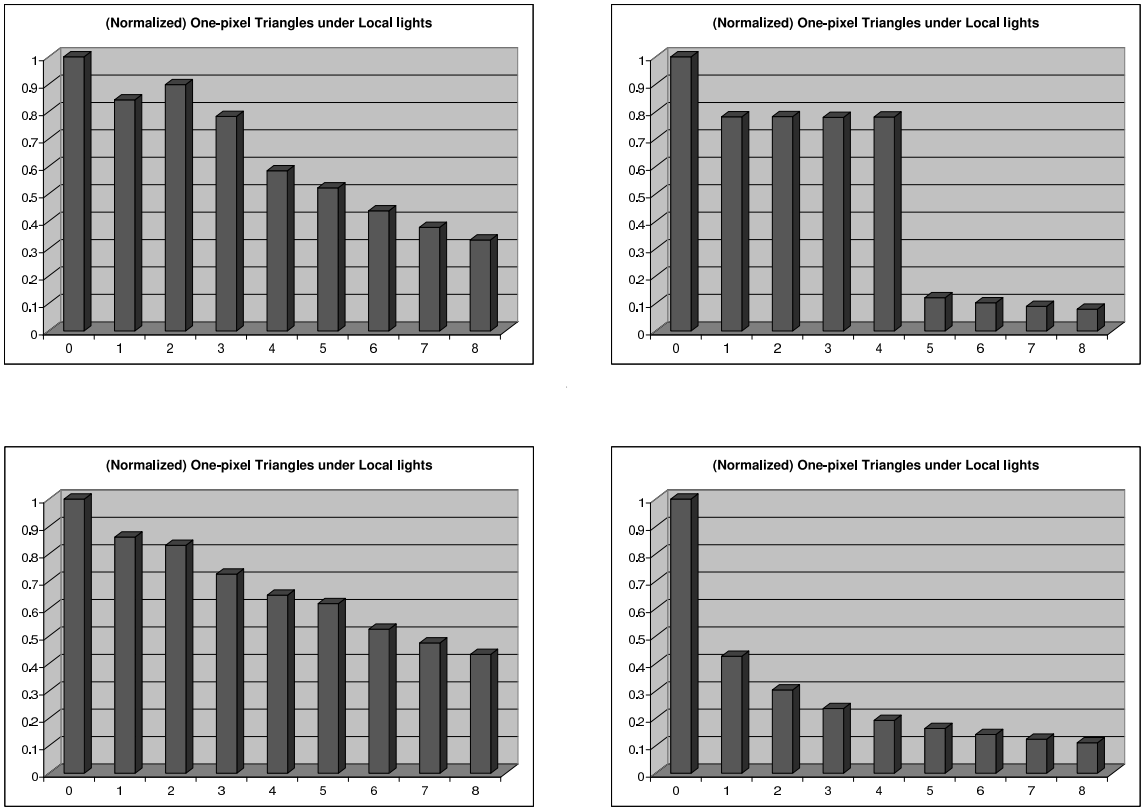

Figure 16: Results for local (point) lights illuminating one-pixel sized triangles

For reference we also present the identical tests with directional-light illumination of the triangles. The difference in the computation required for local vs. directional lights is small.

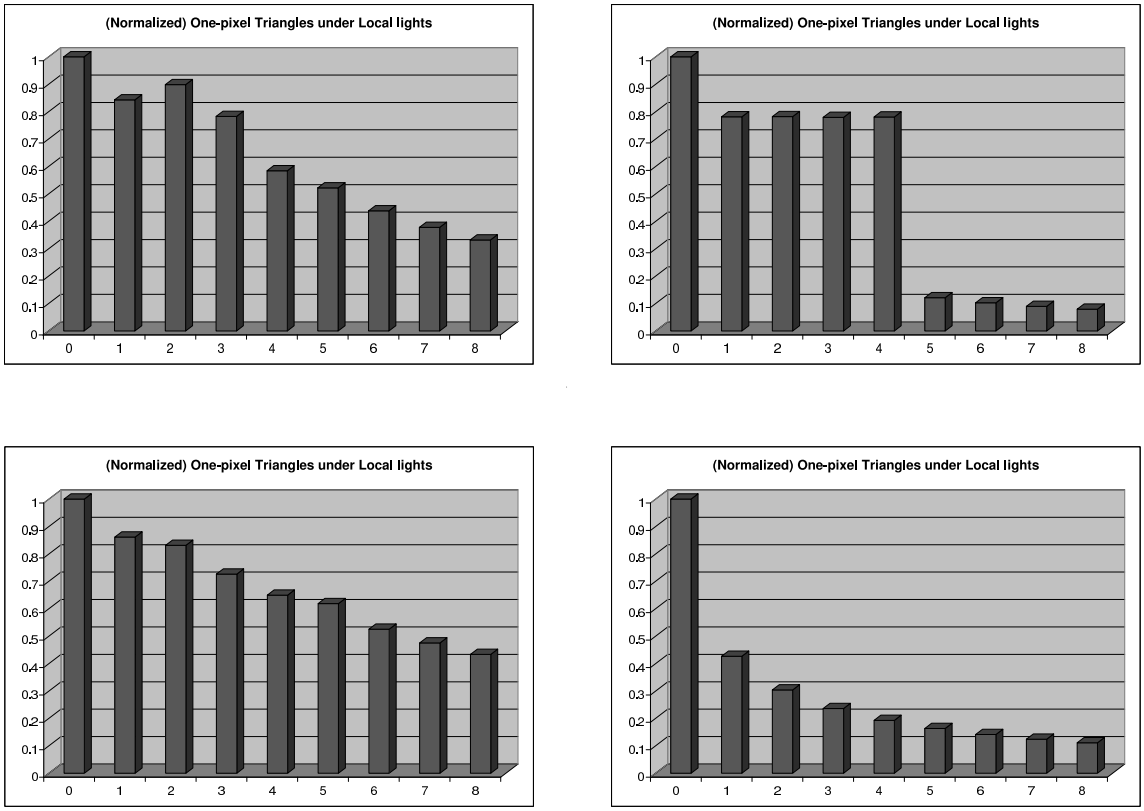

Figure 17: Results for infinite (directional) lights illuminating one-pixel sized triangles

Use connected primitives . . . or don't

Every vertex is transformed, clipped, and perspective divided on its way to becoming a fragment. For many application geometry objects, a single vertex may be shared among many of its primitives, and as such, minimizing the number of times that that vertex is transformed is one way to reduce the load on the transformation side of the OpenGL pipeline. OpenGL has a number of connected primitives that are ideally suited for reducing the number of times that a vertex needs to be transformed and using these primitives can greatly increase performance.

However, connected primitives aren't the answer to all transform limited applications. Not all geometries lend themselves to connected primitives. Consider rendering a cube, which has eight vertices, and six faces. There is no way that the cube can be rendered as a single, continuous connected primitive that maintains its vertex winding properly [3]. This requires that at least two `glBegin()`s are issued for a single cube.

When a rendering command, such as any in Table 1, is issued OpenGL may need to reconfigure its internal rendering pipeline. This reconfiguration is called *validation*. In particular, for `glBegin()`, when the type of geometric primitive (e.g., `GL_POINTS`, `GL_LINES`, etc.) is changed from the previous call to `glBegin()`, the routines used for rasterizing those primitives need to be reconfigured and will most likely invoke a validation.

[3] Technically, this statement really depends on how much of an OpenGL purist one chooses to be. If rendering zero-area triangles is acceptable (which is more of a personal taste issue than a technical one), then a cube can indeed be rendered as a single triangle strip with zero-area triangles being added to change the winding of the tri-strip progression

`glAccum()`	`glBegin()`
`glTexImage1D()`	`glTexSubImage1D()`
`glTexImage2D()`	`glTexSubImage2D()`
`glTexImage3D()`	`glTexSubImage3D()`
`glDrawPixels()`	`glCopyPixels()`
`glReadPixels()`	

(The above list is not exhaustive; routines defined in extensions are not included).

Table 1: OpenGL rendering commands that invoke a validation

Validation: keeping state consistent To an OpenGL application author OpenGL is a simple state machine with two operational states: setting state, and rendering utilizing that state. The following state diagram describes the situation:

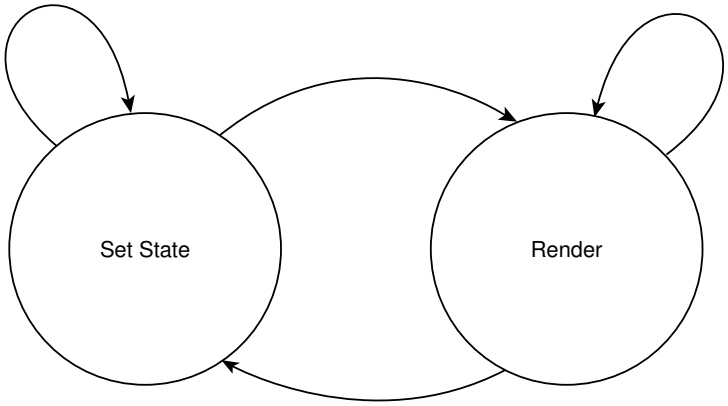

Figure 18: An application programmer's view of the OpenGL state machine

However, this model lacks an important step: validation. Validation is the operation that OpenGL utilizes to keep its internal state consistent with what the application has requested. Additionally, OpenGL uses the validation phase as an opportunity to update its internal caches and function pointers to process rendering requests appropriately. Adding validation to the state diagram from Figure 19 would look like:

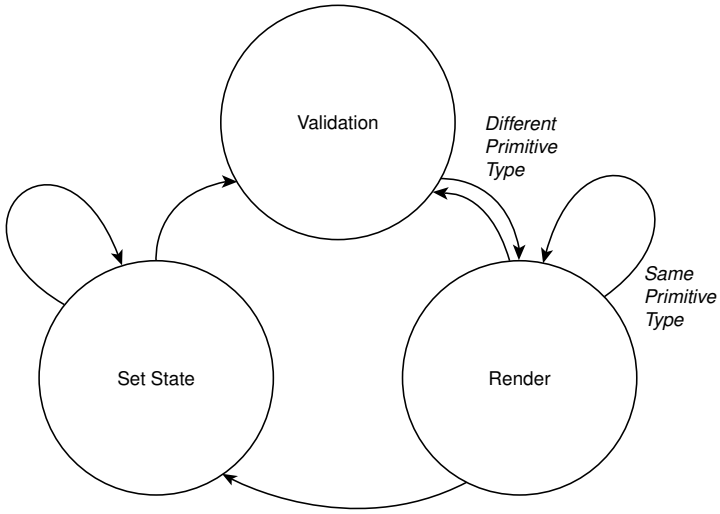

Figure 19: OpenGL state machine taking into consideration the validation phase

Example 3 code sequence invoking an validation at next `glBegin()`

```
glBegin( GL_TRIANGLES );
  ...
glEnd();

glPolygonStipple( stippleMask );
glEnable( GL_POLYGON_STIPPLE );

glBegin( GL_TRIANGLES );
  ...
glEnd();
```

As shown in Example 3, when the `glPolygonStipple()` call is made, OpenGL copies the stipple mask to its internal state, and marks an internal flag indicating that state has changed, and a validation is requested at the next rendering state change. Likewise, when the polygon stipple is enabled with the `glEnable()` call, validation is again requested at the next rendering state change.

When the `glBegin()` is finally executed, the validation that was requested multiple times is performed and the parts of the OpenGL machine that were affected by the state changes are updated. In the stipple example, the rasterizer for polygons is switched from the default one to the stippled rasterizer.

For a more thorough indication of the affects of validation, consider the following simple case study.

You need to render a large number (say 10,000) of statically positioned cubes in 3D space.

For simplicity's sake, assume that the vertices of each cube have already been transformed to their necessary location. There are a number of ways to approach the problem:

1. Perhaps you create a routine that takes an array of the eight vertex positions for a single cube, such as

Example 4 rendering a cube as six quads

```
void drawCube( GLfloat color[], GLfloat* vertex[] )
{
  static int  idx[6][4] = {
    { ... },
  };

  glColor3fv( color );
  glBegin( GL_QUADS );
  for ( int i = 0; i < 6; ++i )
    for ( int j = 0; j < 4; ++j )
      glVertex3fv( vertex[idx[i][j]] );
  glEnd();
}
```

where `drawCube()` is called in a loop to render each cube.

Ignoring issues related to loop overhead and host-memory access, this method suffices, but you notice that you're duplicating a considerable amount of effort. Each vertex of the cube is passed to the OpenGL pipeline three separate times which means you are doing the same exact work three times for the same vertex.

2. Upon reconsideration of the wasted efforts of transforming vertices, you decide to use connected primitives to minimize the number of vertex transformations. Rewriting `drawCube()` slightly,

Example 5 rendering a cube as two quads and a quad-strip

```
void drawCube( GLfloat color[], GLfloat* vertex[] )
{
  static int  idx[6][4] = {
    { ... },
  };

  glColor3fv( color );
  glBegin( GL_QUADS );
  for ( int i = 0; i < 4; ++i )  // Render top of cube
    glVertex3fv( vertex[idx[0][i] );
  for ( i = 0; i < 4; ++i )  // Render bottom of cube
    glVertex3fv( vertex[idx[1][i] );
  glEnd();

  glBegin( GL_QUAD_STRIP );
  for ( i = 2; i < 6; ++i ) {
    glVertex3fv( vertex[idx[i][0]] );
    glVertex3fv( vertex[idx[i][1]] );
  }
  glVertex3fv( vertex[idx[2][0]] );
  glVertex3fv( vertex[idx[2][1]] );
  glEnd();
}
```

In this case, you've reduced the number of vertex transformations from 24 to 14, saving 140,000 vertex transformations per frame.

3. Feeling inspired, you rewrite `drawCube()` once more, this time, moving the `glBegin()` and `glEnd()` outside of `drawCube()` and the loop in the main application, as in

Example 6 rendering a cube as six quads without a `glBegin()`/`glEnd()` pair in the base routine

```
void drawCube( GLfloat color[], GLfloat* vertex[] )
{
  static int  idx[6][4] = {
    { ... },
  };

  glColor3fv( color );
  for ( int i = 0; i < 6; ++i )
    for ( int j = 0; j < 4; ++j )
      glVertex3fv( verts[idx[i][j]] );
}
```

and in the main application rendering loop:

Example 7 outer loop used with Example 4

```
glBegin( GL_QUADS );
for ( int i = 0; i < numCubes; ++i )
  drawCube( color[i], vertex[8*i] );
glEnd();
```

The results of conducting this test on three different OpenGL implementations yielded the following normalized results.

Code Technique	Machine A	Machine B	Machine C
Example 5	3.99	4.36	2.23
Example 4	3.99	4.50	2.15
Example 7	1.00	1.00	1.00

Table 2: Normalized results for various computing platforms

For each of the architectures Example 7 is by far the fastest, even though a considerable number of additional vertices are being transformed each frame. Some of the overhead is due to function calls (there are 10,000 less `glBegin()` and `glEnd()` calls being made in Example 7), but most can be attributed to state changing and validation.

These tests were conducted using only *immediate mode* rendering. As OpenGL supports four methods for transferring vertex data to the rendering pipeline: immediate mode, *display lists*, *vertex arrays*, and *interleaved arrays*, another reasonable test would be to determine which method of passing geometry is the best for each platform of interest.

Returning to our case study from before, we can now explain why transforming 33% more vertices went faster than the seemingly more optimized case utilizing connected primitives. This simple example illustrates the affect

that validation can have on an OpenGL application and suggests that minimizing validations can also improve the performance of the application.

More on minimizing validation, particularly with respect to state changes, will be discussed in the "State Sorting" section.

The lure of object-oriented programming Object-oriented programming is a tremendous step in the quality of software engineering. Analyzing the larger problem as a set of smaller problems with interactions is a great aid to better application design. Unfortunately, object-oriented programming's "encapsulation" paradigm can cause significant performance problems if one chooses the obvious implementation for rendering geometric objects.

For instance, one obvious encapsulation of a cube-type object is illustrated in Example 8:

Example 8 a sensible C++ class that encapsulates a geometric object that could invoke more validations than are truly necessary

```
class Cube : public GeometricObject {

  enum { X = 0, Y, Z };

  float  position[3];
  float  color[3];
  float  angle;   // angle and axis for orientation
  float  axis[3];
  float  scale[3];

  static float  vertex[8][3] = { ... };

  virtual void render( void ) {
    glPushMatrix();
    glTranslatef( position[X], position[Y], position[Z] );
    glRotatef( angle, axis[X], axis[Y], axis[Z] );
    glScalef( scale[X], scale[Y], scale[Z] );
    glColor3fv( color );

    // Render cube in some manner: GL_QUADS, GL_QUAD_STRIP, etc.

    glPopMatrix();
  }
};
```

Even though this is a fairly standard, well-written solution to the problem, the encapsulation paradigm hinders tuning your application when considering validation and other techniques of a more global nature. This encapsulation of rendering state, data, and functionality is easily captured in a C++ class as shown in this example, but is truly language independent. That is, most complicated applications will at some point add layers of abstraction to help with code understandability and maintenance. These abstraction layers need to be thought about carefully in the context of the full OpenGL pipeline in order to ensure that performance is not lost as the application increases in complexity. Thoughtful abstraction can yield performance gains as state can be managed more precisely, objects sorted in more efficient ways, data managed more efficiently, and more.

One solution to this problem, which has multiple benefits, is to utilize the *Visitor Pattern*, as described in [4]. Of course, one could employ visitor patterns with a class structure similar to the one's described above. In addition to employing the visitor pattern, making the separation of OpenGL state, and rendering data is essential. This topic is addressed in the "State Sorting" section.

Determining the best way to pass geometry to the pipe

OpenGL has five ways of transmitting geometric data to the OpenGL pipeline: immediate mode, display lists, vertex arrays, interleaved arrays, and vertex buffer objects (VBOs). Although all five methods are perfectly legal OpenGL commands, their performance across various platforms varies greatly, usually based on the systems memory architecture (e.g., bus speed, use of DMA, etc.).

Immediate mode is the easiest mode to use and is what most application developers use. There are two major performance disadvantages to immediate mode. First, immediate mode geometry requires a large number of function calls. Therefore, immediate mode may not be the fastest mode of execution on computing architectures where function call overhead is expensive. However, it may also be the case that the other modes of rendering utilize immediate mode calls in their internals. Secondly, immediate mode rendering transmits all the vertex data for an application's geometry across the graphics bus. Modern graphics cards can process vertices significantly faster than they can be fed across the graphics bus (even AGP 8x), so immediate mode almost guarantees being bottlenecked at the bus to the graphics device.

Display lists collect most OpenGL calls (exceptions generally represent calls that return a value `glGet*()`, `glGenLists()`, etc.). Depending upon the hardware architecture and driver implementation display lists may merely be large allocated arrays of host memory that collect and tokenize the OpenGL input and then replay that input upon display list execution. Optimization of the token stream may occur but does not happen in some drivers. If the stream is optimized, then, as an example, vertex data may be stored in memory on the graphics device, and so using display lists can yield tremendous performance benefits.

Vertex arrays are a method of specifying all of the geometric data to the OpenGL pipe in a few function calls with hopes that the driver can efficiently process this data. This approach was principally designed to alleviate the function call overhead problem. The base implementation of the calls for rendering vertex arrays: `glDrawArrays()`, `glDrawElements()`, `glArrayElement()`, and `glDrawRangeElements()` is permitted to be simply the appropriate execution of immediate mode calls, or may be a more hardware appropriate optimized method. One additional consideration for vertex arrays is memory access for the host processor. As the data for the vertices is distributed among multiple arrays, memory access may affect the execution performance of vertex arrays.

Interleaved vertex arrays are a variant of vertex arrays, with the characteristic that the data for each vertex is stored in a contiguous block of memory. For example, am array of C language "struct" for a single vertex often can be mapped into an interleaved vertex array, as illustrated in Example 9. For certain architectures this may be the best format for specifying geometry — the only way to know is to conduct a benchmark case. Interleaved arrays can be specified using both `glInterleavedArrays()` using one of the predefined vertex formats, or using the ordinary vertex array functions with pointers into the interleaved data and an appropriate stride.

Example 9 Pseudocode demonstrating setup and rendering of interleaved arrays

```
typedef struct VertexStruct {
    float texture[2];
    float color[4];
    float normal[3];
    float coord[3];
} Vertex;

Vertex vertices[] = {/* application fills in vertices */};
int vertexCount = /* number of vertices in array */;

/* Set up the vertex array */
glInterleavedArrays(GL_T2F_C4F_N3F_V3F, sizeof(VertexStruct), vertices);

/* Draw the entire array as triangles */
glDrawArrays(GL_TRIANGLES, 0, vertexCount);
```

Both immediate mode and vertex arrays transmit the vertex data across the graphics bus every frame. You might note that OpenGL has the vertex array pointers, so why would it not copy the vertex data down to the graphics device at array specification time? Vertex arrays were designed primarily to reduce functional call overhead, and the semantics of vertex arrays allow applications to alter the data in vertex arrays at any time!

The ”vertex buffer object” extension to OpenGL 1.4 and core feature in OpenGL 1.5 were designed to allow vertex arrays to be created directly in graphics device memory, and then specified through the normal vertex array functions, including interleaved vertex arrays. The authors consider this to be the fastest geometry specification technique an application can use, if the feature is available. Instead of providing a pointer to, for example, `glVertexPointer()`, the application first specifies a current ”buffer” with `glBindBuffer()`, and then provides an offset within that buffer to `glVertexPointer()` instead of a pointer to memory.

Finally, many modern device accelerators contain a ”vertex cache”, which caches some small number (e.g., 12 to 24) of post-vertex-program transformed vertices. In order to take best advantage of this vertex cache, an application would use indexed primitives. Rather than providing vertex data which corresponds one-to-one with the vertices issued for the geometry, the application provides an index for each vertex in the geometry, indicating which entry in the vertex array should be used. To store this data in device memory, the index array is specified through an ”element” array buffer object, then referenced through an offset to `glDrawElements()`. This is illustrated using the `GL_ARB_vertex_buffer_object` extension in Example 10, but looks almost identical in OpenGL 1.5. Finally, the index list may be reordered to better use the cache. There are publicly available tools for reordering an index list to optimize for cache performance.

Example 10 Setup and rendering of vertex buffer objects and element array object

```
#define BUFFER_OFFSET(n) ((char *)NULL + n)

typedef struct VertexStruct {
    float texture[2];
    float color[4];
    float normal[3];
    float coord[3];
} Vertex;

Vertex vertices[] = {/* application fills in vertices */};
int vertexCount = /* number of vertices in array */;

/* ideally, index data is optimized for vertex reuse through vertex cache */
uint indices[] = {/* application fills in indices */};
int indexCount;

int vertexBuffer;
int elementBuffer;

glGenBuffersARB(1, &vertexBuffer);
glBindBufferARB(GL_ARRAY_BUFFER_ARB, vertexBuffer);
glBufferDataARB(GL_ARRAY_BUFFER_ARB, sizeof(Vertex) * vertexCount,
    vertices, GL_STATIC_DRAW_ARB);

glGenBuffersARB(1, &indexBuffer);
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, indexBuffer);
glBufferDataARB(GL_ELEMENT_ARRAY_BUFFER_ARB, sizeof(uint) * indexCount,
    indices, GL_STATIC_DRAW_ARB);

/* Set up the vertex array, could be portion of vertex buffer but */
/* for this example we specify offset of 0 */
glInterleavedArrays(GL_T2F_C4F_N3F_V3F, sizeof(VertexStruct), BUFFER_OFFSET(0));

/* Draw the entire array as triangles */
glDrawElements(GL_TRIANGLES, indexCount, GL_UNSIGNED_INT, BUFFER_OFFSET(0));
```

Use OpenGL transformation routines

Another simple way to increase OpenGL's transformation performance is to utilize OpenGL's transformation routines: `glTranslate*()`, `glRotate*()`, `glScale*()`, `glOrtho()`, and `glLoadIdentity()`. These routines have a benefit over the more generic routines `glMultMatrix()`, and `glLoadMatrix()`. At worst, the "typed" routines will cause a full matrix multiply, just as could be predicted. However, for optimized versions of OpenGL, which includes most software renderers, OpenGL tracks changes to the current matrix (the matrix at the top of each matrix stack). OpenGL does this to try and minimize the computation required for a transformation.

When OpenGL is first initialized (at context creation time), each matrix stack is loaded with an identity matrix. As changes are made to the matrices by matrix multiplication and loading matrices, each change is tracked. Some properties that are tracked include:

- realizing that the w-coordinate column is of the form $(0\ 0\ 0\ 1)^T$, as in the case of rotations and scales
- determining that the matrix is a 2D rotation, as in a rotation around the z axis
- verifying that the matrix is a 2D non-rotational matrix, as in the case of a scale
- knowing when the matrix is an identity
- realizing that the transformation is specifying screen coordinates (this mostly is for the benefit of the `GL_PROJECTION` matrix)

Each of the situations above reduce the number of multiplications and additions required for the transformations as compared to a generic 4×4 matrix. The matrix type is changed whenever a transformation type call is made. In the case of the generic matrix routines (which in this case, include `glFrustum()`, as the matrix generated in this case doesn't yield may optimizations), regardless of what type matrix is currently at the top of the matrix stack, the type is changed to the generic type, and a full matrix multiplication is done for each vertex transformation. Although in limited instances one may be able to determine what type a generic matrix is, this is not commonly done due to floating-point comparison issues.

An optimization example: Terrain mesh

The customer's application

Here's an example of a program which the authors have optimized using many of the techniques previously described.

The customer's data is a large two-dimensional array of height field data representing terrain, 4001 samples on one side by 4801 samples on the other. The customer found that rendering the height field at its original resolution as triangles (4000 by 4800 quads, each represented by two triangles) was too slow, so the application downsampled the data by a factor of 16 in each direction, thus rendering only 1/256th the number of triangles in the original. This yielded 150,000 triangles per frame at ten frames per second on a 2003-era workstation.

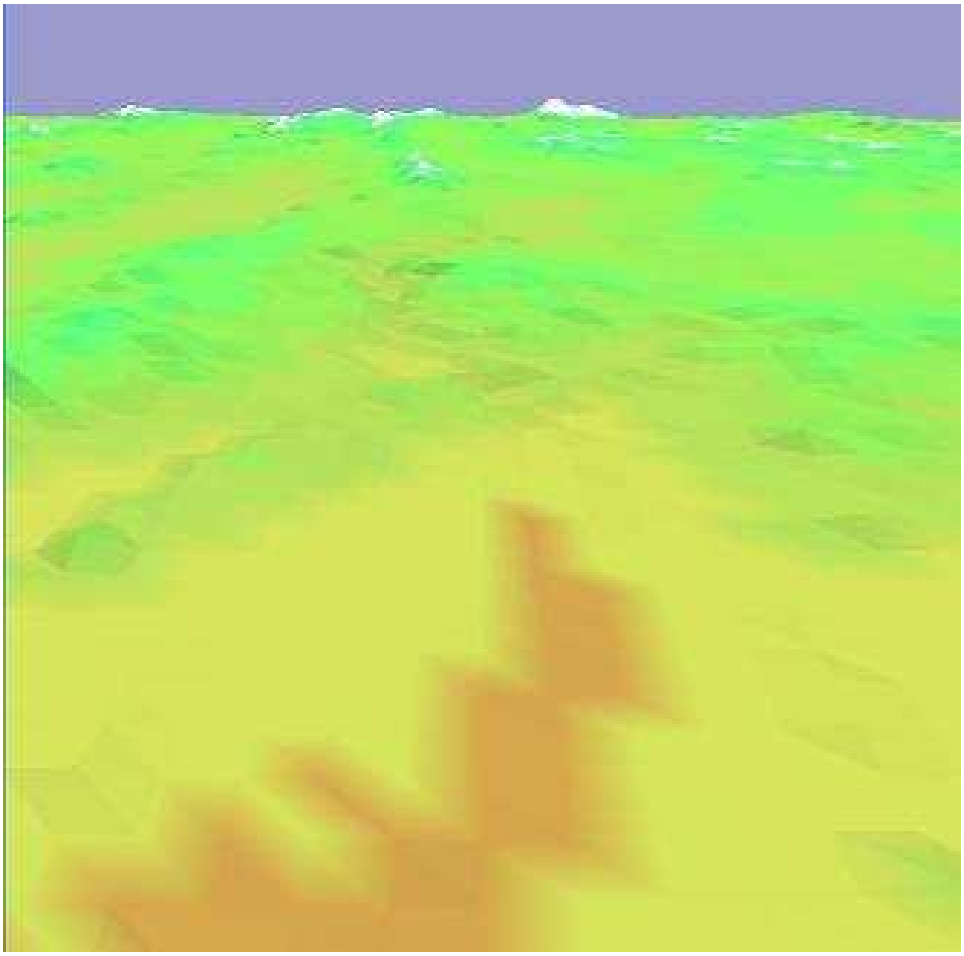

Figure 20: Original meshed terrain from heightfield

The application rendered each triangle in immediate mode, generating the vertex coordinate, color, and normal for each triangle each frame.

Example 11 Customer application code for specifying heightfield color

```
const GLfloat color0[3]  = { 0.65, 0.40, 0.10 };
const GLfloat color1[3]  = { 0.60, 0.50, 0.15 };
const GLfloat color2[3]  = { 0.65, 0.55, 0.25 };
const GLfloat color3[3]  = { 0.70, 0.55, 0.25 };
const GLfloat color4[3]  = { 0.70, 0.75, 0.30 };
const GLfloat color5[3]  = { 0.60, 0.75, 0.30 };
const GLfloat color6[3]  = { 0.50, 0.80, 0.30 };
const GLfloat color7[3]  = { 0.40, 0.85, 0.35 };
const GLfloat color8[3]  = { 0.30, 0.85, 0.45 };
const GLfloat color9[3]  = { 0.80, 0.80, 0.80 };
const GLfloat color10[3] = { 1.00, 1.00, 1.00 };

if (elev < 0.0)          glColor3fv (color0);
else if (elev < 304.8)   glColor3fv (color1);
else if (elev < 609.6)   glColor3fv (color2);
else if (elev < 914.4)   glColor3fv (color3);
else if (elev < 1219.2)  glColor3fv (color4);
else if (elev < 1524.0)  glColor3fv (color5);
else if (elev < 1828.8)  glColor3fv (color6);
else if (elev < 2133.6)  glColor3fv (color7);
else if (elev < 2438.4)  glColor3fv (color8);
else if (elev < 2743.2)  glColor3fv (color9);
else                     glColor3fv (color10);
```

This is perfectly valid OpenGL code. Without even considering whether the application was close to peak speeds, note that not only is the data being generated unnecessarily each frame, but that the use of the immediate mode API limits the application to the speed of the bus from the CPU to the graphics device.

Identifying Performance Goals

An application programmer's first optimization step should be to identify an acceptable performance level, and determine whether an application is approaching or exceeding that level.

This particular workstation was capable of rendering 100 million vertices per second from memory on the graphics device. More importantly, though, for this application, the connection from the system to the graphics device was an AGP 4x bus. Each vertex in the generated data contained 9 floats, 3 each for coordinate, color, and normal, totalling 36 bytes. Since an AGP 4x bus can theoretically transmit approximately 1 gigabyte per second, we can calculate that this application could never exceed 27.7 million vertices per second in immediate mode. Perhaps 50% of that bus-transfer peak would be considered acceptable, or 13.4 million vertices per second. We use this as our primary goal.

Secondarily, though, it would be nice to approach the device's peak, 100 million vertices per second. We use 50% of this number as our secondary goal.

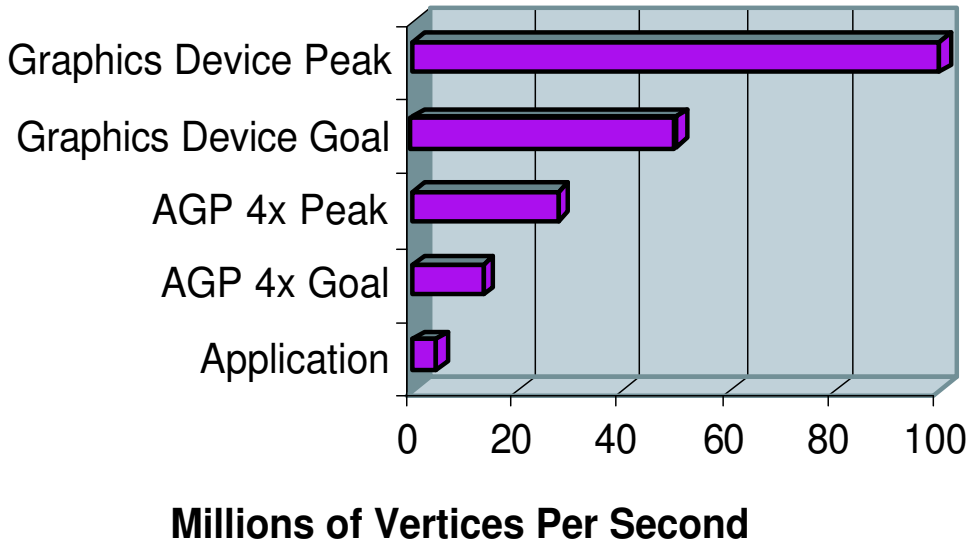

Figure 21: Vertex Performance

The application rendered 150,000 triangles per frame at ten frames per second, or 4.5 million vertices per second. We can see right away that the application was only reaching 32% of the speed we have established is our primary goal, and only 9% of the secondary goal.

Identifying and Widening the Bottleneck

Next, we try to identify the bottleneck. Resizing the rendering window made no difference, so it was clear that the application was not fill-limited. Replacing `glVertex()` calls with `glNormal()` calls also made no difference, so the application was not transform-limited. Normally, the next step would be to store the geometry in a display list or vertex array object and test whether transporting the geometry was an issue. Because the data was being generated every frame, however, the generation code had to be altered to run only once at application initialization time, and that makes it difficult to say whether the problem was the generation of the data or the download. Instead, the per-frame generation of the data was converted to a one-time initialization step without altering the immediate mode download method. This yielded an immediate improvement and exceeded the primary performance goal, at the cost of the memory used to store the vertex data.

In some sense, the customer could have stopped there, depending on his/her project plan. The frame rate was improved, but the customer could also have downsampled at a higher resolution to simply display more data at the same frame rate as the original application. This would be a perfectly good place to stop, since the primary performance goal was reached. We're stubborn, though, and wanted to squeeze the customer's money's worth out of the workstation. So we applied a few more techniques.

More Techniques for Performance

Although the color data was being stored (and downloaded) in three floats, it's clear that a vertex's color in this application is dependent only on the height of the vertex. We can use `glTexGen()` with `GL_LINEAR` texture coordinate generation mode, and map the vertex height into a 1D texture, instead of storing and downloading 12 bytes per vertex of color data. This reduces our memory consumption and increases the rate at which we can send vertices. Additionally, interpolating a texture coordinate across a polygon can provide many color transitions, while interpolating between two vertex colors provides less visual information.

Example 12 Code for providing heightfield color through glTexGen

```
init() {

    ... // other OpenGL init

    float texgenPlane[4];
    unsigned int texobj;

    int colorMapEntryCount;
    unsigned char colorMap[...];
    float colormapMinElev;
    float colormapMaxElev;

    // Create color map, containing "colorMapEntryCount" entries (power of
    // two, of course), mapping color to elevations between "colormapMinElev"
    // and "colormapMaxElev".

    texgenPlane[0] = 0.0;
    texgenPlane[1] = 1 / (colorMapMaxElev - colorMapMinElev);
    texgenPlane[2] = 0.0;
    texgenPlane[3] = colorMapMinElev / (colorMapMaxElev - colorMapMinElev);

    glGenTextures(1, &texobj);
    glBindTexture(GL_TEXTURE_1D, texobj);
    glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, colorMapEntryCount, 0,
        GL_RGB, GL_FLOAT, colorMap);
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
}

drawFrame() {
    // clear, set matrices, etc

    // draw some annotation, etc

    glBindTexture(GL_TEXTURE_1D, texobj);
    glEnable(GL_TEXTURE_1D);
    glEnable(GL_TEXTURE_GEN_S);
    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
    glTexGenfv(GL_S, GL_OBJECT_PLANE, texgenPlane);

    // draw terrain mesh

    // draw more annotation, etc
    // swapbuffers, etc
}
```

Next, in order to improve geometry rate even further, we created a vertex buffer object in order to store the

terrain data directly on the graphics device. This is the most direct way to approach the device's peak raw speed, and using the glTexGen technique in the previous paragraph means we can store more vertices in device memory as well.

As mentioned previously, using indexed data allows geometry to reuse the results of previously transformed vertices. So we used short indexed triangle strips in spans, so that each strip reused some of the transformed vertices from the previous strip. This allowed us to exceed our secondary goal.

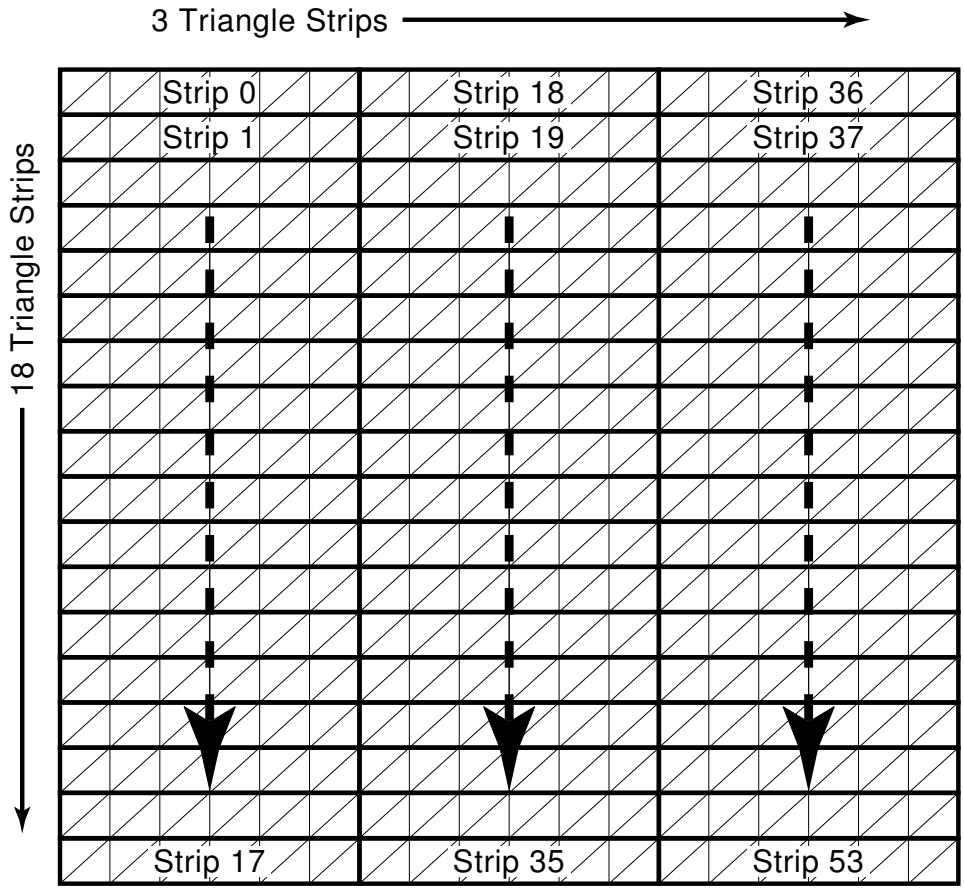

Figure 22: Improving transformed vertex reuse by optimizing for the vertex cache

Finally, we noted that the application was often used to zoom in to a section of the terrain. Since portions of the terrain were then off-screen, we diced the terrain into tiles and implemented view-frustum culling, so that tiles not inside the view frustum were simply not drawn. Our implementation of this technique used the occlusion query extension for ease of implementation, but occlusion query is not required.

figure of view frustum and tiles culled and not culled

Because vertex buffer objects are paged in and out of device memory as necessary, this also gave us an opportunity to create more terrain tiles than could fit on the device, and allow frustum culling to page tiles in and out automatically. When the visible set of tiles all fit in device memory, then the tiles ran nearly at peak speed for the device.

In the final implementation of this terrain viewing application, the data set was rendered at a resolution of

2001 samples per side, or fifty times higher resolution than the original downsampled data. The lowest frame rate measured was 3 frames per second for the entire data set, or 24 million vertices per second, nearly twice our primary goal. However, the frame rate for viewing a portion of the data set was usually 10 frames per second or better, and often 60 frames per second.

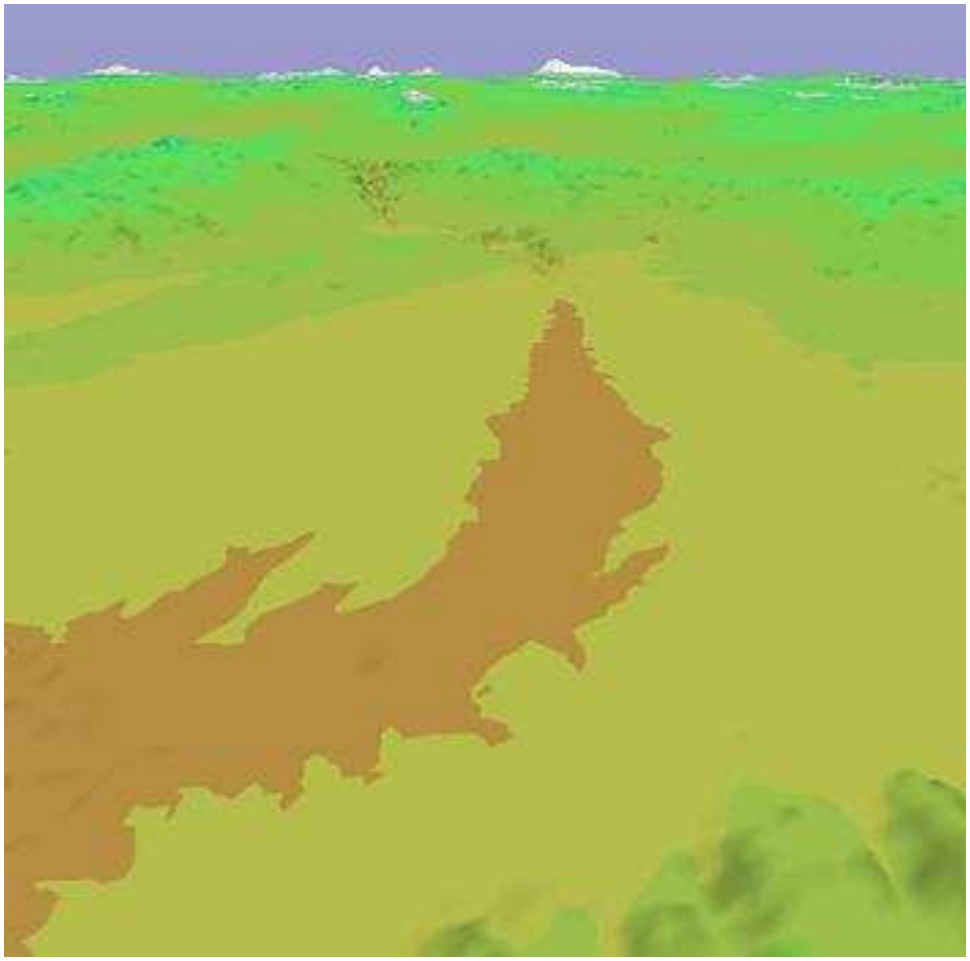

Figure 23: Final optimized terrain

General OpenGL Performance Techniques

State Sorting - Trying to organize rendering around state changes

Each time certain state values (excluding vertex data such as color, normals, etc.) are updated, OpenGL prepares to do a validation process at the next rendering operation. A logical optimization step is then to try and minimize the number of times that state needs to be set. For instance, if the application needs to draw two objects with blue materials, and two with red, the more efficient (and hopefully logical) progression would to render both objects of one color first, then the other set, requiring only two material changes per frame. This technique is generally referred to as "state sorting" and attempts to organize rendering requests based around the types of state that will need to be updated to get the correct results.

Objects can have considerably different states, and determining a precedence for which states to sort for first is recommended. Generally, the goal is to attempt to sort the render requests and state settings based upon the penalty for setting that particular part of the OpenGL state. For example, loading a new texture map is most likely a considerably more intensive task than setting the diffuse material color, so attempt to group objects based on which texture maps they use, and then make the other state modifications to complete the rendering of the objects.

Use sensible pixel and texture formats

When sending pixel type data down to the OpenGL pipeline try to use pixel formats that closely match either the format of the framebuffer or requested internal texture format. When pixel data is transmitted to the pipe it proceeds through the pixel processing pipeline as illustrated below:

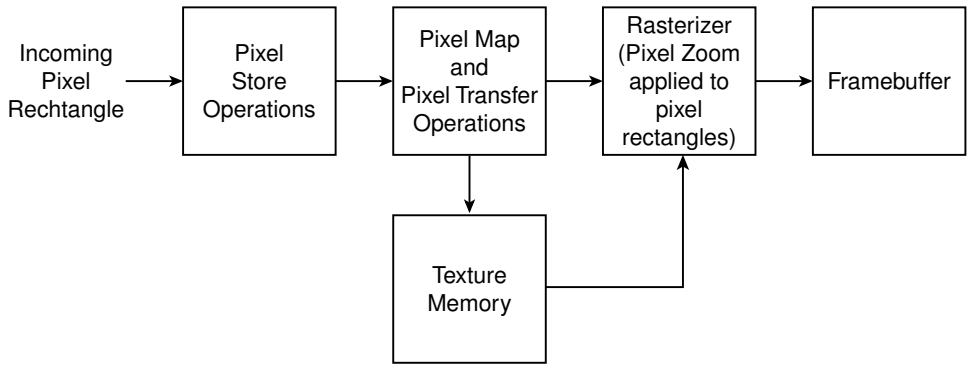

Figure 24: OpenGL's pixel conversion and transfer pipeline

The conversion of pixel data from the format and type specified from the application to format and type that is most palatable to OpenGL happens automatically, if required. This conversion could take a while to complete depending upon the resolution, format, and number of components, . Minimizing these conversion is a good idea as these conversions are both out of the application's direct control and potentially expensive. The following set of graphs represent a comparison of transfer and processing speeds for identical 1024×1024 sized images passed to OpenGL using the `glDrawPixels()` routine. The values were computed based on number of bytes of data, and not pixels, as to attempt to keep the volume of data transferred identical.

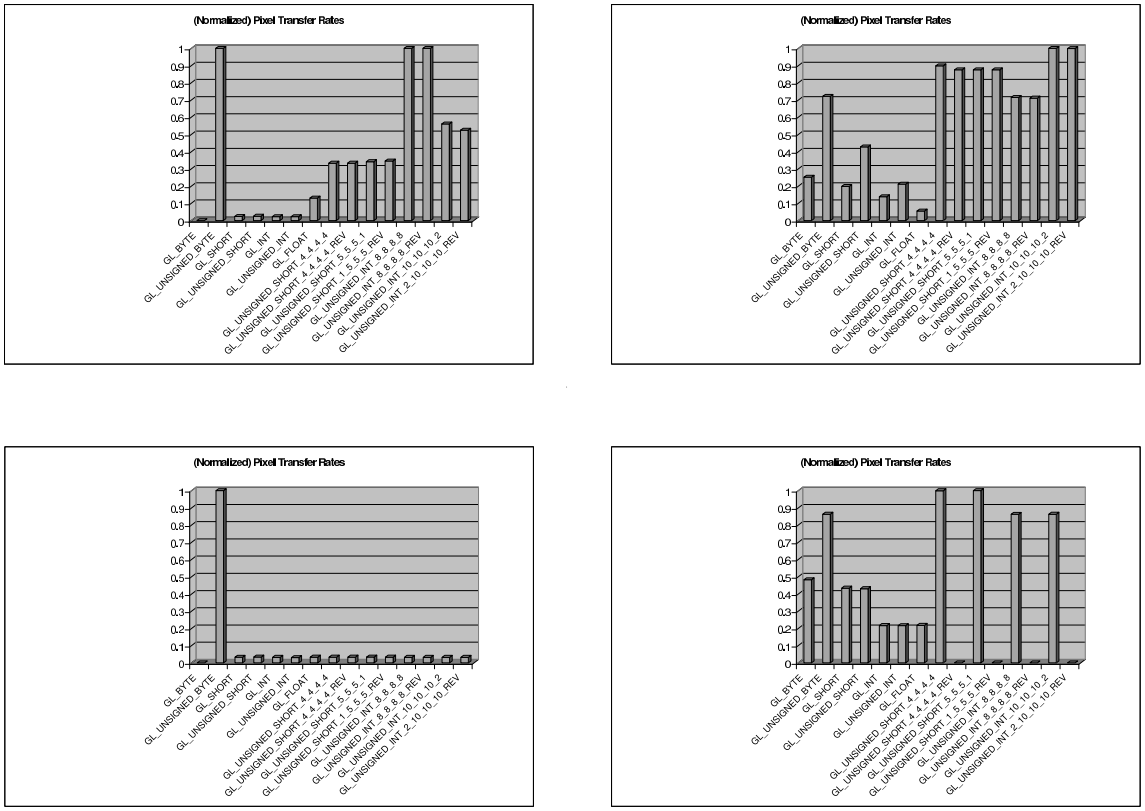

Figure 25: Results for passing different pixel formats into `glDrawPixels()`

It appears, from these empirical results, that `GL_UNSIGNED_BYTE` or a *packed-pixel* format may well be the best choices for sending pixel data into the OpenGL pipeline. These results also provide important considerations for texture image download, as the texel in a texture are processed in the same manner (up to conversion into the texture's internal format. Pixels going to the screen are converted into screen format) as pixels passed down as images to be rendered.

As a general rule of thumb, packed pixels formats (e.g., `GL_UNSIGNED_SHORT_5_5_5_1`), or combinations of type (e.g., `GL_UNSIGNED_BYTE`) and format (e.g., `GL_RGBA`) that match the framebuffer's format will skip conversion. The same is true for texture formats when compared to the texture's internal format. Currently, floating point formats are the least widely supported formats for any hardware available today. This is changing, however with the fully-floating-point graphics hardware beginning to become widely available.

In instances where only a small part of an image is of interest, using `glPixelStore()` is one way to access the pixels of interest. However, this approach may cause avoidable overhead due to memory access on the host machine, and in particular, if there are only incremental changes to the area of interest from frame to frame (as in an image roaming application, for example), it may be more optimal to utilize texture memory as an "image cache" where only the new pixels of the image are updated. See the *Reload textures using `glTexSubImage*D()`* section for more detail.

Pre-transform static objects

For objects that are permanently positioned in world coordinates pre-transforming the coordinates of such objects as compared to calling `glTranslate*()` or other modeling transforms can represent a performance advantage for state management and some geometry-limited scenes.

Use texture objects

In the revision of OpenGL from version 1.0 to version 1.1, a new concept for managing textures was added: texture objects. Although this change happened years ago, reiterating the use of texture objects is useful. Without texture objects, applications that used multiple textures per frame were required to reload each texture per frame causing large numbers of texels to be transferred each frame, or mosaic the textures into a single larger texture, if the texture could be accommodated in texture memory. By utilizing texture objects, management of texture memory, along with down-loading the texels into texture memory was decreased to merely specifying to OpenGL which texture object should be made active. In such a case that all textures fit into texture memory, changing texture maps became almost a free operation.

An additional benefit of texture objects is the ability to add a "priority" to each texture object, providing OpenGL a hint of which textures are the most important to keep resident in texture memory.

Use texture proxies to verify that a given texture map will fit into texture memory

Along with texture objects, texture proxies were added into OpenGL 1.1. This feature provides much more information than merely querying the `GL_MAX_TEXTURE_SIZE`, which specifies the "largest" texture that OpenGL can handle. This value doesn't consider number of components, texel depth, texture dimensionality, or mipmap levels, so it is of very limited use.

Texture proxies, on the other hand, verify that all values for a specific texture: width, height, depth, internal format, and components, are valid and that there is enough texture memory to accommodate the texture, including mipmap levels, if requested.

Reload textures using `glTexSubImage*D()`

For applications that need to update texels in a texture, or refresh an entire texture, it usually is more beneficial to "subload" a texture using `glTexSubImage*D()`, as compared to `glTexImage*D()`. Calls to `glTexImage*D()` request for a texture to be allocated, and if there is a current texture present, deallocate it. Although under limited circumstances (like all parameters of the two textures being identical), the deallocation / allocation phase may be skipped by certain OpenGL drivers, it's unlikely this optimization is present in many implementations.

A better approach is to allocate a texture large enough to hold whatever texels may be required, and load the texture by calling `glTexSubImage*D()`. If the textures are not the same size, the texture matrix can be used to manipulate the texture coordinates to match the new size. By sub-loading the texture, the texture never needs to deallocated. As `glTexSubImage*D()` doesn't permit changes to the texture's internal format, requesting a texture of a different internal format does require a deallocation / allocation phase.

Vertex and Fragment Programming Performance Considerations

Introduction

Traditional graphics hardware has been implemented as fixed-function pipeline. ln recent years, programmable graphics hardware has become available to the extent that all new graphics subsystems expose low-level programmability. This programmability, currently, comes in the form of assembly language-like programs that replace portions of the traditional fixed-function pipeline. These programs are implemented by a common set of base instructions combined with some specific fragment and vertex instructions. These will be described more fully later in the notes.

Programmable Pipeline versus Fixed Function

Programmability in current graphics hardware is provided by two assembly code-like programming facilities. (These are commonly called ”shaders”, but this is slightly inaccurate.)

The first is the *vertex program* that replaces the traditional OpenGL fixed-function transform pipeline as illustrated in Figure 26. It provides the programmer complete control of vertex transformation, lighting, texture-coordinate generation, and allows any other computations to occur on a per-vertex basis. This permits the implementation of custom transformations and lighting equations and the ability to offload these operations from the CPU to the vertex processor in the GPU.

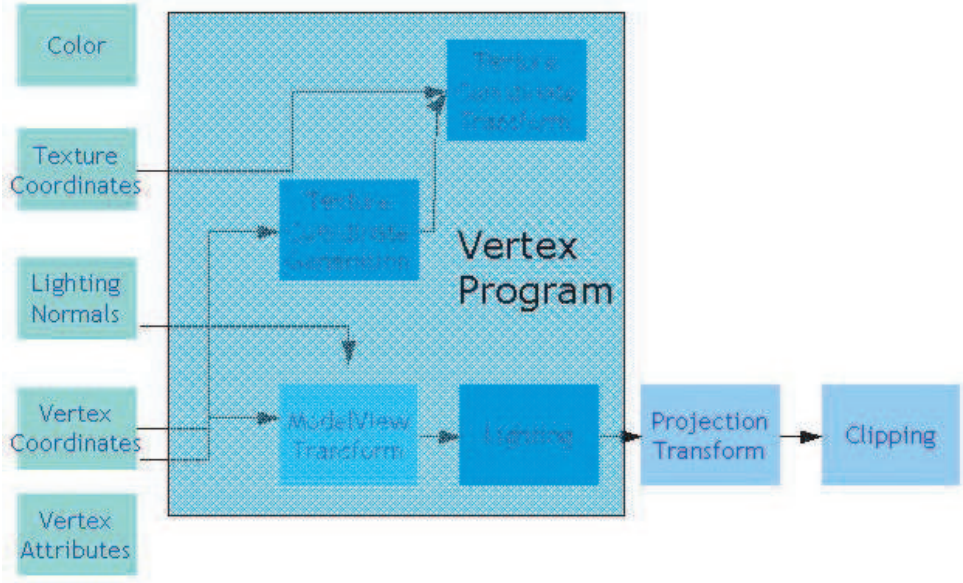

Figure 26: Programmable Vertex Shader Pipeline

The second program is the *fragment program* that replaces the traditional OpenGL fixed-function rasterization pipeline as illustrated in Figure 27. The fragment program provides control of fragment generation by replacing the traditional fixed-function texture blend, color sum, and fog operations. Per-pixel tests (i.e., depth, stencil, scissor, etc.) are still part of the fixed function pipeline. As with the vertex program, the intent behind this programmability is to allow custom blending, color calculations, etc. to allow better lighting, bump-mapping, or other effects not available in fixed-function pipelines.

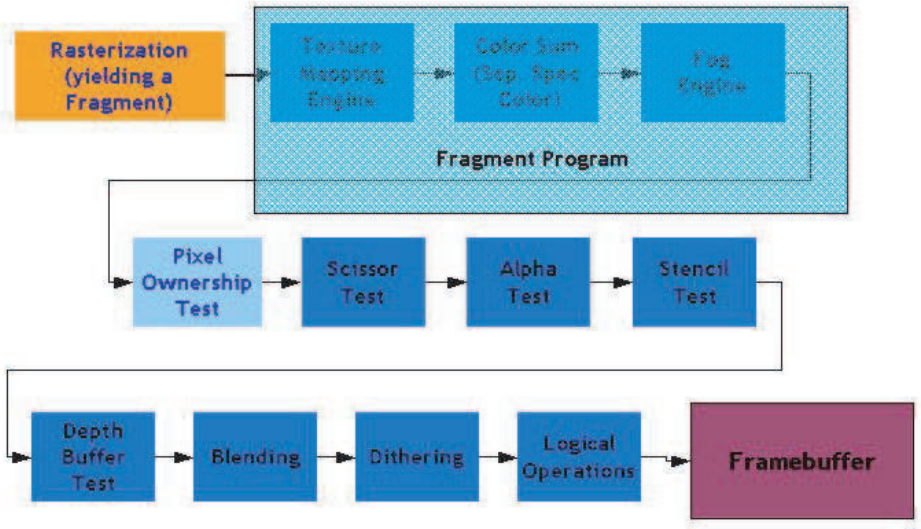

Figure 27: Programmable Fragment Shader Pipeline

Capabilities and Responsibilities

Vertex and fragment programming provide increased flexibility by giving the programmer more complete complete control of transform, lighting, and shading. But with this new capability comes responsibility. In the case of the vertex program, using a programmable shader to replace any part of the fixed function pipeline requires the program to implement all transformation, lighting, and texture-coordinate generation even if it simply mimics what is already implemented in the fixed-function pipeline. Said differently, using a vertex program requires you to implement any or all parts of the fixed-function pipeline that you would like to utilize. Operations incumbent upon the programmer when writing vertex programs are:

- Model-view and Projection Vertex Transformations
- Normal Transformation, Rescaling and Normalization
- Color Material
- Per-Vertex Lighting
- Texture Coordinate Generation
- Texture Matrix Transformations
- Per-Vertex Point Size
- Fog Coordinate Computations
- User Clip Planes

Operations not replaced by the vertex program include:

- Evaluators
- View Frustum Clipping
- Perspective Divide

- Viewport Transformation
- Depth Range Transformation
- Front and Back Color Selection
- Clamping of Primary and Secondary Color
- Primitive Assembly and Setup

In the case of the fragment program it is the responsibility of the programmer to implement all operations that compute the final fragment color prior to per-pixel testing. These operations include:

- Texturing
- Color sum
- Fog

Operations such as

- Polygon-offset depth displacement
- Depth testing
- Stencil testing
- Alpha blending
- Alpha rejection tests

are still processed by the fixed-function fragment pipeline.

SIMD Programming Concepts and Caveats

One key design element in the operation of vertex and fragment programs is that neither has any knowledge of neighboring pixels or vertices. This design makes the vertex and fragment engines Single Instruction Multiple Data (SIMD) machines. This simply means that the same program operates on a batch of data at once. Beneath this design abstraction, actual graphics hardware may process several vertices or fragments in parallel, and then pass those results to the next phases of the pipe. This parallelism enables high-performance graphic accelerators, but with some tradeoffs. For example, a fragment shader would be a great place to try to perform some blending among adjacent fragments, to achieve some sort of smoothing or blur effect. However, to achieve this, you would first have to render to a texture map, then render a quad utilizing that texture map, and apply the fragment program there. In such a case, each texel would correspond at some level to a pixel, and operations could access adjacent texels and blend the results. The result of this SIMD operation means that multi-pass effects are required for programs that require information about adjacent primitives. As multi-pass is a relatively expensive operation, care must be taken when designing programs that use adjacent pixels. In many cases, if the adjacency computation is required, there may be no performance work-around. Hence, take care in the design phase to consider whether or not adjacency information is absolutely required.

Performance Implications

Although vertex and fragment programs provide additional levels of rendering flexibility, performance can vary dramatically from program to program. Programmability enables hardware acceleration of graphics algorithms previously performed on the CPU. However, vertex and fragment programs can become a bottleneck in application performance. This section will explore the performance impacts of vertex and fragment programs.

Basic Programs: Construction and Performance

Program Basics

Fragment and vertex programs share many of the same interfaces and instructions. This section will discuss the basics of constructing a program, downloading it to the hardware, and the differences between vertex and fragment program instruction sets.

A vertex or fragment program is a sequence of instructions presented to the OpenGL machine as a character array. These are then parsed and compiled, frequently with some optimizations applied, such as dead-code-elimination, and then used by the OpenGL machine to replace some subset of pipeline fixed-functionality. Each of these programs are slightly different due to the different problem domains of the vertex engine and the rasterization engine, but the programs also share much structure. Each has a header, a variable declaration section, a list of instructions, and a footer. For example, a basic fragment program could be:

Example 13 A Basic `GL_ARB_fragment_program` based Fragment Program

```
!!ARBfp1.0                          # header

TEMP tmp;                           # variable declaration

MUL tmp, fragment.color, .5;        # instruction
MOV tmp.a, 1;                       # instruction
MOV result.color, tmp;              # instruction

END                                 # footer
```

In large part, the only difference between the structure of this program and a vertex program would be the header, simply changed to `!!ARBvp1.0()` for vertex programs. The contents of the actual instructions would be different, and the results written would be of a different form as well. The variable declaration section will not be explored in much detail as there are many different types of variables that can be declared and the details behind their initialization is not relevant here. A brief tour of the types of variables is this:

TEMP read/write arbitrary temporary 4-vector variables for use within a program.

ATTRIB read-only state associated with the incoming vertex. Extra attribute arrays may be associated with vertex array data and accessed here. see `glVertexAttrib[123]dARB()`

PARAM state associated with the environment of this program. Also used for read-only constants. Things like light parameters can be read here.

We will discuss differences between vertex and fragment instructions, but not go into detail about specific instructions. For complete details on all elements of program syntax and semantics please see the OpenGL Architecture Review Board specifications for the `GL_ARB_vertex_program`[3] and `GL_ARB_fragment_program`[2] extensions, respectively.

As mentioned in the prior paragraph, each program is largely the same structurally, however the major differences occur in the instructions available to each. Even so, the vast majority of instructions available to fragment and vertex programs are the same. Basic math, such as adding, subtracting, and multiplying, vector math, such as dot-product and cross-product, swizzling, etc. are available to each. A complete list of instructions available to a vertex program are the following, with those unique to vertex programs emphasized.

ABS	absolute value
ADD	add
ARL	*address register load*
DP3	3-component dot product
DP4	4-component dot product
DPH	homogeneous dot product
EX2	exponential base 2
EXP	*exponential base 2 (approximate)*
FLR	floor
FRC	fraction
LG2	logarithm base 2
LIT	compute light coefficients
LOG	*logarithm base 2 (approximate)*
MAD	multiply and add
MIN	minimum
MOV	move
MUL	multiply
POW	exponentiate
RCP	reciprocal
RSQ	reciprocal square root
SGE	set on greater than or equal
SLT	set on less than
SUB	subtract
SWZ	extended swizzle
XPD	cross product

Instructions available to a fragment program are the following, with unique instructions emphasized.

ABS	absolute value
ADD	add
CMP	compare
COS	*cosine with reduction to the interval* $[-\pi, \pi]$
DP3	3-component dot product
DP4	4-component dot product
DPH	homogeneous dot product
DST	distance vector
EX2	exponential base 2
FLR	floor
FRC	fraction
KIL	*kill fragment*
LG2	logarithm base 2
LIT	compute light coefficients
LRP	*linear interpolation*
MAD	multiply and add
MAX	maximum
MIN	minimum
MOV	move
MUL	multiply
POW	exponentiate
RCP	reciprocal
RSQ	reciprocal square root
SCS	sine/cosine without reduction
SGE	set on greater than or equal
SIN	*sine with reduction to the interval* $[-\pi, \pi]$
SLT	set on less than
SUB	subtract
SWZ	extended swizzle
TEX	*texture sample*
TXB	*texture sample with bias*
TXP	*texture sample with projection*
XPD	cross product

The major difference between the vertex and fragment program instruction sets is that fragment programs deal with textures, and so have an associated set of instructions largely for manipulation of textures.

The major difference between vertex and fragment programs is simply their frequency of execution. Vertex programs are executed once per-vertex and their results interpolated across the resulting primitive. These results are then passed to a fragment program that is executed once per-fragment in that resulting primitive. Depending on the average size of triangles in your particular shaded object, the ratio between number of vertices to number of fragments can range from 1 to 10s to perhaps millions of pixels (for primitives spanning the entire screen area.) Understanding this ratio becomes key to balancing performance when either vertex or fragment programs become a bottleneck. We'll examine balancing workload between fragment, vertex, and host later in the course.

Program Binding, Compiling, and Usage

Once you have a vertex or a fragment program as a string, downloading it to the graphics hardware and managing it is analogous to managing texture ids. First, begin by calling `glGenProgramsARB()` to get the ID values for the vertex and fragment programs you would like to download. Then, `glBindProgramARB()` using a newly-generated ID. Finally, download the program with `glProgramStringARB()`. Henceforth, that program may be made active by again calling `glBindProgramARB()` with the same ID. Although in most cases error checking is an undesirable action to perform at run-time, shaders require it to validate that the code

you input was correctly parsed and is usable. Call `glGetIntegerv(GL_PROGRAM_ERROR_POSITION_ARB, errpos)` and `glGetString(GL_PROGRAM_ERROR_STRING_ARB)` recursively to find any errors that may be in the downloaded program. When the `errpos()` value returned from `glGetIntegerv()` is `-1()` the code has compiled correctly. A visual method to check code correctness is to see if your object with a fragment or vertex program is black or white—these colors are frequently applied in an error state.

Performance Impact of Common Instructions

The overall performance of a vertex or fragment program is based upon the performance of instructions which compose the program. Each instruction has different performance characteristics depending upon the type of operation that is performed by the instruction. Instruction performance is measured in the number of clock cycles that an instruction takes to execute. Some instructions execute in a single clock while others execute in multiple clock cycles. While cycle counts for those instructions that perform mathematical operations are typically straightforward to predict, the performance of those that perform texture lookups is harder to calculate due to the fact that the instruction time will be based on the overall OpenGL texture state. For example, texture lookup performance will depend upon the texture format—data not in the native texture format for the hardware will require swizzling and masking and these operations will negatively impact texture lookup performance. In all cases, it is a good idea to consult vendor documentation to determine the performance of shader instructions. Figure 28 illustrates the wide variation in instruction performance.

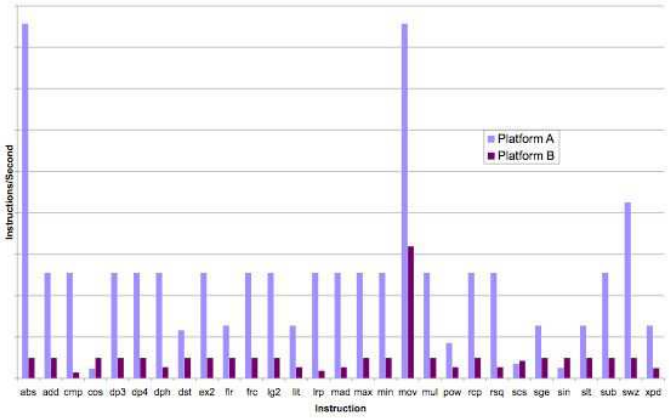

Figure 28: Execution Time for Common Instructions

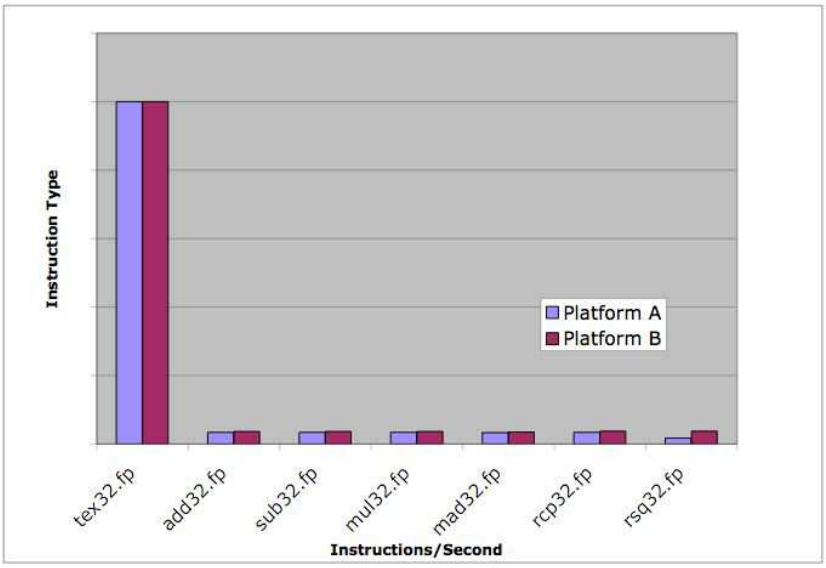

Figure 29: Performance of Texture Instructions compared with ALU Instructions

As illustrated in Figure 29 texture instructions are an order of magnitude faster than ALU instructions. Given that textures can be used as general 1D, 2D and 3D arrays, table lookups via a texture can be used to replace calculations and ultimately improve program performance. Textures also permit dependent lookups (i.e., `a = x[y[i]]`)

Texture coordinates may also be used as general interpolants. In this case, general per-vertex program data can be stored in texture coordinates and subsequently interpolated into per-fragment data during rasterization. This technique is useful for efficiently implementing advanced rendering algorithms that require additional per-fragment attributes.

```
ATTRIB normal   = fragment.texcoord[3];
ATTRIB pressure = fragment.texcoord[2];
```

However, one must exercise caution when using texture lookups in this way as texture sampling may lead to interpolation and continuity problems.

Longer programs tend to require more execution time due to the fact that additional instructions require additional cycles. However, making a program shorter does not guarantee that it will execute in fewer cycles. The order of instructions within vertex and fragment programs can also play a role in the overall performance of a program. Again, vendor documentation is the best guide here. One potential way to ensure optimal instruction ordering is to utilize a higher-level shading language and rely on the vendor's compiler to optimize the order of the resulting program instruction.

When coding a vertex or fragment program, it is best to code up the program and ensure that it is working properly prior to optimization. In general, program performance is difficult to predict based on program contents. This is due to the fact that:

1. the code looks like assembly, but in reality it is actually compiled for a specific platform

2. the native hardware instruction set is different

3. because programs are compiled, dead code may be removed and instructions may be reordered

All of these can impact performance. Some manual code optimizations may make a program take more cycles to execute, when compared with the built-in translation and optimization. As a result, trial and error is often the best approach to optimization. The next section will discuss shader program performance analysis and optimization in more detail.

Analysis and Optimization

In this section we will discuss a variety of tools and techniques for evaluating program performance, both on an individual program level, and from a rendering system perspective.

As we have seen earlier in the notes, programs that are graphics bound are either transform or fill limited. As is obvious, these two bottlenecks correspond nicely to vertex and fragment programs. If an application using vertex or fragment programs is, for example, fragment limited, a variety of options exist for optimizing that code directly. First among these options is to simply evaluate what operations are being performed, and if less-expensive operations can be used instead. For example, replacing a complex set of math operations, with a pre-calculated set of results, looked up in a texture. Another technique to reduce the number of operations would be to pack them more tightly, to re-use results, to calculate fewer lights, octaves of noise, etc. Similar options exist for vertex programs, where calculations could be perhaps pre-calculated on the host, and passed in as attribute data, etc.

A second major way in which vertex and fragment program bottlenecks can be mitigated is by shifting computational load up and down the pipeline. For example, if an application is transform limited, some of the computation done per-vertex could be done per-fragment, and thereby relieve some of the burden on the vertex engine. Alternately, that work could be pushed up-stream to the CPU, and passed down as attribute data. If fill-limited, computations could be done in a vertex program, and passed down to a fragment program as texture-coordinates, interpolated automatically across each primitive. Utilizing the programmability of each portion of the pipeline to balance workload among all portions is a very powerful technique for managing graphics bottlenecks.

State Issues

Vertex and fragment programs, like any other OpenGL state, have attendant costs in changing those state values. So, as with any other state, care must be taken to change state only as frequently as necessary. But vertex and fragment programs are different than most other OpenGL state in that they rely heavily on the other portions of the pipe to be configured properly. For example, a fragment shader may use a texture unit to provide per-fragment normal values to an object. For this fragment program to work properly, that texture must be loaded into the texture unit specified by that program. Similarly, for a vertex program to operate properly, it may require certain additional data to be loaded in its attribute arrays, so that vertex program can't work in just a generic sense; it requires data fed to it to be formatted a certain way. Finally vertex and fragment programs are often closely coupled, as the output of the Vertex program frequently will compute data in a precise way (say, texture unit 3 is an interpolated binormal) for a fragment program.

Because of the implicit state required by a vertex or fragment program, changing them frequently can be much more expensive than any other form of state, as they require configuration of multiple state elements, such as texture units, lighting configurations, vertex data formats, etc. In order to maximize the overall performance of an application with multiple vertex and fragment programs in it, the programs should be resident for as long as possible. Said differently, the number of primitives rendered per-program-bind should be as great as possible. Switch infrequently, render lots of data in between switches, and you'll be started down the path to good aggregate vertex/fragment program usage and performance.

References

[1] ARB and Dave Shreiner, editors. *The OpenGL Reference Manual*. Addison-Wesley, fourth edition, 2004.

[2] The OpenGL Architecture Review Board. *The* `GL_ARB_fragment_program` *specification*. The OpenGL Architecture Review Board, 2003. `http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt`.

[3] The OpenGL Architecture Review Board. *The* `GL_ARB_vertex_program` *specification*. The OpenGL Architecture Review Board, 2003. `http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt`.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] Silicon Graphics Computer Systems Inc. The OpenGL Sample Implementation. `http://oss.sgi.com/projects/ogl-sample`.

[6] Mark Segal, Kurt Akeley, and Jon Leech. *The OpenGL Graphics System: A Specification (Version 1.4)*. Silicon Graphics Computer Systems Inc., 2003.

[7] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *The OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley, fourth edition, 2003.

Glossary

binary space partition (BSP) tree A technique of computing occluding geometry by subdividing space using a geometric plane to divide 3D space into two half-spaces. The benefit of BSP trees is that they can be used to determine an appropriate rendering order to minimize depth-buffering, as well as proving a convenient structure for culling geometry from a rendered scene.

performance bottleneck The execution path of an application that is the limiting factor in the performance of the program. Every program has a bottleneck; the issue is if the impact of the bottleneck is severe enough to warrant tuning the application to remove the bottleneck.

display list rendering A mode of OpenGL command execution where OpenGL commands are stored for later execution. Certain hardware implementations of OpenGL may include specialized hardware for executing display lists more rapidly than in immediate mode. An additional benefit to display lists is that when they are executed (by a call to `glCallList()` or `glCallLists()`), only that data needs to be transferred to the rendering server in distributed cases.

fast path Term used to describe a sequences of OpenGL commands that execute hardware accelerated, or in an optimized manner.

fragment Data relevant to a pixel in the framebuffer, including its position, color, texture coordinates, and depth.

infinite (or directional) light A type of OpenGL light that radiates light in a single direction, with all light rays considered parallel. This type of light is generally used to simulate sunlight.

immediate mode rendering A mode of OpenGL command execution where an OpenGL command is executed immediately, as compared to having delayed execution as with display lists.

interleaved array rendering A mode of OpenGL command execution where all vertex data (including colors, normals, texture coordinates, and edge flags) are stored in a single array in an interleaved format such that all the data for a particular vertex is grouped in a particular format (see the documentation for

`glInterleavedArrays()` for a list of accepted formats). This method may produce faster rendering than immediate mode in a similar manner than vertex arrays.

packed pixel formats A storage method utilized by OpenGL to compactly store the color components of an image's pixel in a suitable single data value, as compared to storing each color component in its own data value location. For example, the `GL_UNSIGNED_SHORT_5_5_5_1` stores the red, green, blue, and alpha components in a single unsigned short value, using only two bytes. If packed pixels were not used, this data would need at least four bytes (assuming that unsigned bytes are sufficient pixel format can be found that matches the internal format used by the graphics hardware.

painters' algorithm A technique of sorting (non-intersecting) geometric primitives based on distance from the eye. In such a configuration, the geometry does not need to be depth buffered. The algorithm takes its name from how painters render their scenes, progressing from the background to the foreground objects by painting over objects that are considered farther from the viewer than the new objects.

point (or local) light A type of OpenGL light that radiates light in a spherical manner from a single point, as a light bulb would in the real world. Point lights can also be converted into OpenGL spot lights.

rasterization phase The stage of the OpenGL rendering pipeline that render fragments into color pixels in the framebuffer. This phase includes the following operations: viewport culling, scissor box, depth buffering, alpha testing, stencil testing, accumulation buffer operations, logical pixel operations, dithering, and blending.

shading The process of computing the color of a pixel. Shading can combine colors provided by sampling a texture map, blending with colors already in the framebuffer, and colors modified by anti-aliasing effects.

state sorting The process of organizing OpenGL rendering such that the number of state changes is minimized.

transformation phase The stage of the OpenGL rendering pipeline that transforms vertices in world coordinates into fragments in screen coordinates. This phase includes the following operations: model-view and projection transformations, lighting, texture coordinate generation, per-vertex fog computation, clipping (including user-defined clip planes).

validation The process by which OpenGL initializes its internal state to match the requests of the user. This might include computing intermediate cached results, updating the pipeline function pointers, and other operations.

vertex array rendering A mode of OpenGL rendering where all vertex data (including colors, normals, texture coordinates, and edge flags) is passed to OpenGL as a set of separate arrays of information, for possible batch processing, but mostly to minimize the fine-grained function call overhead that is required for immediate mode rendering.

Updates

These notes are a living document and will be updated with new techniques and further elaborations over time. The latest copy of the notes can be found at

`http://www.PerformanceOpenGL.com/`

Appendix A

The following tables show the different combinations of state that were utilized in generating Figure 3 through Figure 12. The '■' represets the state being enabled, while the '×' represents the particular feature being disabled.

Test Id	2D Texture Mapping	Alpha Blending	Depth Testing	Stencil Testing	Dithering	Alpha Testing	Percentage of Peak Fill	Percentage Change
1	×	×	×	■	■	×	100.00%	
2	×	×	×	×	■	■	99.99%	0.01%
3	×	×	×	■	■	■	99.97%	0.02%
4	×	×	×	■	×	■	99.97%	0.00%
5	×	×	×	×	■	×	99.96%	0.01%
6	×	×	×	×	×	■	99.94%	0.02%
7	×	×	×	■	×	×	99.92%	0.02%
8	×	×	×	×	×	×	99.86%	0.07%
9	×	×	■	×	■	■	99.23%	0.62%
10	×	×	■	■	×	■	99.20%	0.04%
11	×	×	■	×	×	■	99.18%	0.02%
12	×	×	■	×	■	×	99.16%	0.01%
13	×	×	■	■	×	×	99.14%	0.02%
14	×	×	■	■	■	■	99.14%	0.00%
15	×	×	■	■	■	×	99.12%	0.02%
16	×	×	■	×	×	×	98.78%	0.34%
17	×	■	×	×	×	■	76.39%	22.67%
18	×	■	×	■	×	■	76.38%	0.01%
19	×	■	×	■	×	×	76.38%	0.00%
20	×	■	×	×	×	×	76.38%	0.01%
21	×	■	×	■	■	■	76.37%	0.00%
22	×	■	×	×	■	■	76.37%	0.00%
23	×	■	×	■	■	×	76.36%	0.01%
24	×	■	×	×	■	×	76.33%	0.04%
25	×	■	■	■	×	■	70.69%	7.39%
26	×	■	■	■	×	×	70.69%	0.00%
27	×	■	■	■	■	×	70.69%	0.01%
28	×	■	■	×	×	■	70.68%	0.00%
29	×	■	■	×	■	■	70.68%	0.01%
30	×	■	■	×	■	×	70.68%	0.00%
31	×	■	■	■	■	■	70.67%	0.00%
32	×	■	■	×	×	×	70.67%	0.01%
33	■	×	×	×	■	■	59.27%	16.13%
34	■	×	×	■	×	■	59.26%	0.02%
35	■	×	×	■	×	×	59.25%	0.01%
36	■	×	×	×	■	×	59.24%	0.02%
37	■	×	×	×	×	×	59.24%	0.00%
38	■	×	×	■	■	×	59.24%	0.01%
39	■	×	×	×	×	■	59.23%	0.01%
40	■	×	×	■	■	■	59.22%	0.03%
41	■	×	■	■	■	■	55.90%	5.61%
42	■	×	■	×	■	■	55.88%	0.02%
43	■	×	■	×	×	×	55.87%	0.02%
44	■	×	■	■	■	×	55.87%	0.01%
45	■	×	■	■	×	×	55.87%	0.00%
46	■	×	■	■	×	■	55.87%	0.00%
47	■	×	■	×	×	■	55.87%	0.00%
48	■	×	■	×	■	×	55.86%	0.01%
49	■	■	×	×	■	■	49.24%	11.85%
50	■	■	×	■	■	■	49.24%	0.00%
51	■	■	×	■	×	×	49.23%	0.03%
52	■	■	×	×	×	×	49.23%	0.00%
53	■	■	×	■	■	×	49.22%	0.00%
54	■	■	×	×	■	×	49.22%	0.00%
55	■	■	×	×	×	■	49.22%	0.00%
56	■	■	×	■	×	■	49.22%	0.01%
57	■	■	■	■	■	■	47.22%	4.06%
58	■	■	■	×	■	■	47.22%	0.01%
59	■	■	■	■	■	×	47.22%	0.00%
60	■	■	■	×	×	■	47.21%	0.00%
61	■	■	■	■	×	■	47.21%	0.00%
62	■	■	■	×	■	×	47.21%	0.00%
63	■	■	■	×	×	×	47.19%	0.04%
64	■	■	■	■	×	×	47.19%	0.01%

Results for Machine 1

Test Id	2D Texture Mapping	Alpha Blending	Depth Testing	Stencil Testing	Dithering	Alpha Testing	Percentage of Peak Fill	Percentage Change
1	×	×	×	×	×	×	100.00%	
2	×	×	■	■	×	■	100.00%	0.00%
3	×	×	■	×	×	■	99.99%	0.01%
4	×	×	■	■	×	×	99.98%	0.01%
5	×	×	×	×	×	■	99.97%	0.01%
6	×	×	×	■	×	×	99.88%	0.08%
7	×	×	■	×	×	×	99.88%	0.00%
8	×	×	×	■	×	■	95.37%	4.52%
9	×	■	■	×	×	×	68.99%	27.66%
10	×	■	×	×	×	×	68.99%	0.00%
11	×	■	■	×	■	×	68.98%	0.01%
12	×	×	■	×	■	■	68.98%	0.00%
13	×	■	×	■	■	■	68.98%	0.00%
14	×	■	×	■	×	■	68.98%	0.00%
15	×	×	×	■	■	■	68.98%	0.00%
16	×	■	■	■	■	×	68.97%	0.00%
17	×	■	×	■	×	×	68.97%	0.00%
18	×	■	■	×	■	■	68.97%	0.01%
19	×	×	■	■	■	×	68.97%	0.00%
20	×	■	■	■	■	■	68.96%	0.00%
21	×	×	■	×	■	×	68.96%	0.00%
22	×	×	×	×	■	■	68.96%	0.00%
23	×	■	×	×	×	■	68.96%	0.01%
24	×	■	■	■	×	×	68.96%	0.00%
25	×	×	×	■	■	×	68.96%	0.00%
26	×	■	×	×	■	■	68.91%	0.07%
27	×	×	■	■	■	■	68.89%	0.03%
28	×	■	×	×	■	×	68.89%	0.00%
29	×	■	■	×	×	■	68.88%	0.00%
30	×	■	■	■	×	■	68.88%	0.00%
31	×	■	×	■	■	×	68.86%	0.04%
32	×	×	×	×	■	×	68.65%	0.30%
33	■	×	×	×	×	×	45.00%	34.45%
34	■	■	■	×	■	■	44.93%	0.16%
35	■	×	×	■	×	■	44.93%	0.01%
36	■	×	×	×	■	■	44.93%	0.00%
37	■	■	×	×	■	×	44.93%	0.00%
38	■	■	×	■	■	×	44.92%	0.00%
39	■	×	×	×	×	■	44.92%	0.00%
40	■	×	■	×	■	■	44.92%	0.00%
41	■	×	■	■	×	■	44.92%	0.00%
42	■	■	×	×	×	■	44.92%	0.00%
43	■	×	×	■	■	×	44.92%	0.00%
44	■	×	×	×	■	×	44.92%	0.00%
45	■	×	■	×	■	×	44.92%	0.00%
46	■	×	■	×	×	■	44.92%	0.00%
47	■	■	×	■	■	■	44.92%	0.00%
48	■	■	■	■	■	■	44.92%	0.00%
49	■	×	×	■	■	■	44.92%	0.00%
50	■	■	×	×	■	■	44.92%	0.00%
51	■	■	×	×	×	×	44.92%	0.00%
52	■	×	■	■	■	×	44.92%	0.00%
53	■	■	■	■	■	×	44.92%	0.00%
54	■	■	■	×	■	×	44.91%	0.00%
55	■	■	×	■	×	■	44.91%	0.00%
56	■	■	×	■	×	×	44.91%	0.00%
57	■	■	■	×	×	×	44.88%	0.06%
58	■	■	■	■	×	■	44.88%	0.00%
59	■	×	■	×	×	×	44.88%	0.00%
60	■	■	■	×	×	■	44.88%	0.00%
61	■	×	×	■	×	×	44.87%	0.02%
62	■	×	■	■	×	×	44.86%	0.03%
63	■	×	■	■	■	■	44.85%	0.02%
64	■	■	■	■	×	×	44.84%	0.02%

Results for Machine 2

Test Id	2D Texture Mapping	Alpha Blending	Depth Testing	Stencil Testing	Dithering	Alpha Testing	Percentage of Peak Fill	Percentage Change
1	×	×	×	×	■	■	100.00%	
2	×	×	×	×	×	×	99.98%	0.02%
3	×	×	×	×	×	■	99.97%	0.01%
4	×	×	×	×	■	×	99.95%	0.02%
5	■	×	×	×	×	■	98.64%	1.31%
6	■	×	×	×	■	■	98.61%	0.04%
7	■	×	×	×	■	×	98.58%	0.02%
8	■	×	×	×	×	×	98.58%	0.00%
9	×	×	■	×	■	■	85.66%	13.11%
10	×	×	■	×	×	■	85.65%	0.01%
11	×	×	■	×	×	×	85.64%	0.01%
12	×	×	■	×	■	×	85.62%	0.02%
13	×	×	■	■	×	×	85.32%	0.35%
14	×	×	■	■	■	×	85.32%	0.01%
15	×	×	■	■	■	■	85.31%	0.01%
16	×	×	■	■	×	■	85.28%	0.04%
17	■	×	■	■	■	×	85.27%	0.01%
18	■	×	■	■	×	■	85.26%	0.00%
19	■	×	■	■	×	×	85.25%	0.01%
20	■	×	■	■	■	■	85.25%	0.00%
21	×	×	×	■	×	■	83.80%	1.70%
22	×	×	×	■	■	■	83.78%	0.02%
23	×	×	×	■	×	×	83.74%	0.05%
24	×	×	×	■	■	×	83.72%	0.03%
25	■	×	×	■	×	×	83.31%	0.48%
26	■	×	×	■	■	■	83.29%	0.03%
27	■	×	×	■	■	×	83.29%	0.00%
28	■	×	×	■	×	■	83.25%	0.04%
29	■	×	■	×	×	×	82.31%	1.14%
30	■	×	■	×	■	×	82.30%	0.01%
31	■	×	■	×	×	■	82.28%	0.03%
32	■	×	■	×	■	■	82.25%	0.04%
33	×	■	×	×	■	■	72.68%	11.64%
34	×	■	×	×	×	×	72.66%	0.02%
35	×	■	×	×	×	■	72.65%	0.02%
36	×	■	×	×	■	×	72.64%	0.01%
37	×	■	×	■	■	×	71.68%	1.32%
38	×	■	×	■	×	×	71.68%	0.00%
39	×	■	×	■	×	■	71.68%	0.00%
40	×	■	■	■	■	×	71.67%	0.00%
41	×	■	×	■	■	■	71.67%	0.00%
42	×	■	■	■	×	■	71.66%	0.02%
43	×	■	■	■	■	■	71.66%	0.00%
44	×	■	■	■	×	×	71.65%	0.02%
45	■	■	×	■	×	×	69.25%	3.34%
46	■	■	×	■	×	■	69.25%	0.00%
47	■	■	×	×	■	■	69.25%	0.01%
48	■	■	×	■	■	■	69.24%	0.01%
49	■	■	×	×	×	×	69.24%	0.00%
50	■	■	×	■	■	×	69.24%	0.00%
51	■	■	×	×	×	■	69.23%	0.01%
52	■	■	■	■	■	■	69.23%	0.00%
53	■	■	×	×	■	×	69.22%	0.00%
54	■	■	■	■	×	×	69.21%	0.01%
55	■	■	■	■	×	■	69.21%	0.01%
56	■	■	■	■	■	×	69.20%	0.00%
57	×	■	■	×	×	×	67.47%	2.50%
58	×	■	■	×	×	■	67.45%	0.04%
59	×	■	■	×	■	×	67.45%	0.00%
60	×	■	■	×	■	■	67.44%	0.02%
61	■	■	■	×	■	■	64.32%	4.61%
62	■	■	■	×	×	×	64.32%	0.00%
63	■	■	■	×	■	×	64.31%	0.02%
64	■	■	■	×	×	■	64.31%	0.00%

Results for Machine 3

Test Id	2D Texture Mapping	Alpha Blending	Depth Testing	Stencil Testing	Dithering	Alpha Testing	Percentage of Peak Fill	Percentage Change
1	×	×	■	×	■	×	100.00%	
2	×	×	■	×	■	■	99.99%	0.01%
3	×	×	■	■	×	×	99.99%	0.00%
4	×	×	■	×	×	■	99.98%	0.01%
5	×	×	×	■	■	■	99.98%	0.00%
6	×	×	■	×	×	×	99.98%	0.00%
7	×	×	×	■	×	■	99.96%	0.01%
8	×	×	×	×	×	×	99.93%	0.04%
9	×	×	×	×	×	■	99.93%	0.00%
10	×	×	■	■	■	×	99.93%	0.00%
11	×	■	■	■	■	×	99.92%	0.01%
12	×	×	×	×	■	■	99.91%	0.01%
13	×	■	×	■	×	■	99.91%	0.00%
14	×	■	■	■	■	■	99.90%	0.00%
15	×	■	■	×	■	■	99.90%	0.01%
16	×	■	×	■	■	×	99.90%	0.00%
17	×	■	×	×	■	×	99.89%	0.00%
18	×	■	×	×	×	■	99.89%	0.01%
19	×	■	■	×	×	×	99.89%	0.00%
20	×	■	×	×	×	×	99.87%	0.01%
21	×	×	■	■	×	■	99.87%	0.00%
22	×	×	■	■	■	■	99.87%	0.00%
23	×	■	×	■	■	■	99.87%	0.00%
24	×	■	×	×	■	■	99.86%	0.01%
25	×	■	×	■	×	×	99.86%	0.00%
26	×	■	■	×	■	×	99.86%	0.00%
27	×	■	■	×	×	■	99.86%	0.00%
28	×	■	■	■	×	■	99.86%	0.00%
29	×	■	■	■	×	×	99.84%	0.01%
30	×	×	×	■	×	×	99.67%	0.17%
31	×	×	×	×	■	×	99.60%	0.07%
32	×	×	×	■	■	×	99.60%	0.00%
33	■	×	■	×	■	■	54.35%	45.43%
34	■	×	■	×	■	×	54.35%	0.01%
35	■	×	×	■	■	×	54.35%	0.00%
36	■	×	■	×	×	■	54.34%	0.01%
37	■	×	×	×	■	×	54.34%	0.01%
38	■	×	×	■	■	■	54.34%	0.00%
39	■	×	×	■	×	■	54.33%	0.00%
40	■	×	×	×	×	■	54.33%	0.00%
41	■	■	×	×	■	■	54.33%	0.00%
42	■	×	■	×	×	×	54.33%	0.01%
43	■	■	■	■	■	×	54.33%	0.01%
44	■	×	■	■	■	■	54.33%	0.00%
45	■	■	■	■	■	■	54.32%	0.00%
46	■	×	■	■	×	■	54.32%	0.00%
47	■	■	■	■	×	■	54.32%	0.00%
48	■	■	■	×	×	×	54.32%	0.00%
49	■	■	×	■	×	×	54.32%	0.00%
50	■	■	×	×	×	×	54.32%	0.00%
51	■	■	■	×	■	×	54.31%	0.01%
52	■	■	■	■	×	×	54.31%	0.00%
53	■	■	×	■	■	■	54.31%	0.00%
54	■	■	×	■	×	■	54.31%	0.00%
55	■	■	×	×	■	×	54.31%	0.00%
56	■	×	■	■	■	×	54.31%	0.00%
57	■	■	×	×	×	■	54.31%	0.00%
58	■	■	■	×	■	■	54.31%	0.00%
59	■	■	×	■	■	×	54.31%	0.00%
60	■	■	■	×	×	■	54.31%	0.00%
61	■	×	×	×	×	×	54.25%	0.11%
62	■	×	×	■	×	×	54.24%	0.01%
63	■	×	×	×	■	■	54.23%	0.01%
64	■	×	■	■	×	×	54.15%	0.15%

Results for Machine 4

Test Id	2D Texture Mapping	Alpha Blending	Depth Testing	Stencil Testing	Dithering	Alpha Testing	Percentage of Peak Fill	Percentage Change
1	×	×	×	×	■	×	100.00%	
2	×	■	×	×	×	×	99.99%	0.01%
3	×	■	×	×	■	×	99.96%	0.02%
4	×	■	■	×	■	×	99.96%	0.01%
5	×	×	■	×	×	×	99.95%	0.00%
6	×	×	×	×	×	×	99.90%	0.05%
7	×	×	■	×	■	×	99.88%	0.02%
8	×	■	■	×	×	×	99.81%	0.07%
9	×	×	■	×	■	■	99.09%	0.72%
10	×	■	×	×	×	■	99.08%	0.01%
11	×	■	■	×	■	■	99.05%	0.03%
12	×	■	■	×	×	■	99.02%	0.03%
13	×	×	■	×	×	■	99.01%	0.02%
14	×	×	×	×	■	■	98.98%	0.03%
15	×	■	×	×	■	■	98.85%	0.13%
16	×	×	×	×	×	■	98.84%	0.01%
17	×	■	×	■	×	×	77.02%	22.08%
18	×	×	×	■	■	×	77.00%	0.03%
19	×	×	■	■	×	×	76.93%	0.09%
20	×	×	×	■	×	×	76.87%	0.08%
21	×	■	×	■	■	×	76.65%	0.29%
22	×	×	■	■	■	×	76.58%	0.08%
23	×	■	■	■	×	×	76.50%	0.11%
24	×	■	■	■	■	×	76.24%	0.34%
25	×	■	■	■	■	■	70.75%	7.19%
26	×	■	×	■	■	■	70.74%	0.01%
27	×	■	■	■	×	■	70.71%	0.04%
28	×	×	×	■	■	■	70.63%	0.12%
29	×	■	×	■	×	■	70.61%	0.02%
30	×	×	■	■	■	■	70.53%	0.12%
31	×	×	■	■	×	■	70.51%	0.04%
32	×	×	×	■	×	■	70.44%	0.10%
33	■	■	■	×	■	×	58.59%	16.81%
34	■	×	■	×	×	×	58.54%	0.09%
35	■	■	■	×	×	×	58.54%	0.00%
36	■	■	×	×	×	×	58.54%	0.01%
37	■	×	■	×	■	×	58.54%	0.00%
38	■	×	×	×	■	×	58.53%	0.01%
39	■	×	×	×	×	×	58.53%	0.01%
40	■	■	×	×	■	×	58.51%	0.03%
41	■	×	×	×	■	■	55.21%	5.63%
42	■	×	■	×	×	■	55.20%	0.02%
43	■	×	×	×	×	■	55.14%	0.12%
44	■	■	×	×	■	■	55.12%	0.03%
45	■	■	■	×	×	■	55.05%	0.13%
46	■	■	■	×	■	■	55.04%	0.03%
47	■	■	×	×	×	■	55.00%	0.06%
48	■	×	■	×	■	■	54.99%	0.01%
49	■	■	×	■	■	×	48.80%	11.26%
50	■	■	■	■	■	×	48.76%	0.08%
51	■	×	×	■	■	×	48.73%	0.07%
52	■	×	■	■	×	×	48.72%	0.01%
53	■	■	■	■	×	×	48.71%	0.03%
54	■	■	×	■	×	×	48.68%	0.05%
55	■	×	×	■	×	×	48.68%	0.01%
56	■	×	■	■	■	×	48.65%	0.07%
57	■	×	■	■	×	■	46.72%	3.95%
58	■	×	×	■	×	■	46.67%	0.11%
59	■	×	×	■	■	■	46.66%	0.02%
60	■	■	■	■	■	■	46.66%	0.00%
61	■	■	×	■	×	■	46.61%	0.11%
62	■	■	×	■	■	■	46.59%	0.04%
63	■	×	■	■	■	■	46.52%	0.16%
64	■	■	■	■	×	■	46.45%	0.15%

Results for Machine 5

Test Id	2D Texture Mapping	Alpha Blending	Depth Testing	Stencil Testing	Dithering	Alpha Testing	Percentage of Peak Fill	Percentage Change
1	×	■	×	×	×	×	100.00%	
2	×	×	×	×	×	×	99.99%	0.01%
3	×	×	■	×	×	×	99.79%	0.21%
4	×	■	■	×	×	×	99.72%	0.06%
5	×	■	×	×	■	■	99.13%	0.59%
6	×	×	■	×	■	■	99.13%	0.00%
7	×	■	■	×	■	■	99.12%	0.02%
8	×	×	×	×	■	■	99.03%	0.09%
9	×	■	■	×	×	■	98.84%	0.20%
10	×	×	×	×	×	■	98.80%	0.04%
11	×	×	■	×	×	■	98.78%	0.01%
12	×	■	×	×	×	■	98.76%	0.02%
13	×	×	×	×	■	×	86.57%	12.34%
14	×	■	×	×	■	×	86.56%	0.01%
15	×	×	■	×	■	×	86.55%	0.02%
16	×	■	■	×	■	×	86.53%	0.02%
17	×	×	■	■	×	×	78.03%	9.83%
18	×	×	×	■	×	×	78.02%	0.01%
19	×	■	■	■	×	×	77.99%	0.03%
20	×	■	×	■	×	×	77.95%	0.06%
21	×	×	×	■	×	■	71.38%	8.42%
22	×	■	×	■	×	■	71.38%	0.01%
23	×	×	■	■	×	■	71.36%	0.02%
24	×	■	■	■	×	■	71.31%	0.07%
25	×	■	■	■	■	■	70.80%	0.71%
26	×	■	×	■	■	■	70.77%	0.05%
27	×	×	×	■	■	■	70.76%	0.01%
28	×	×	■	■	■	■	70.70%	0.09%
29	■	■	×	×	×	×	60.10%	14.99%
30	■	×	×	×	×	×	60.09%	0.01%
31	×	■	■	■	■	×	59.95%	0.24%
32	■	■	■	×	×	×	59.95%	0.00%
33	×	■	×	■	■	×	59.94%	0.01%
34	×	×	×	■	■	×	59.94%	0.00%
35	×	×	■	■	■	×	59.94%	0.00%
36	■	×	■	×	×	×	59.93%	0.01%
37	■	■	■	×	■	■	57.14%	4.67%
38	■	■	×	×	■	■	57.12%	0.03%
39	■	×	×	×	■	■	57.11%	0.02%
40	■	×	■	×	■	■	57.10%	0.02%
41	■	■	×	×	×	■	56.00%	1.93%
42	■	×	■	×	×	■	55.96%	0.08%
43	■	■	■	×	×	■	55.96%	0.00%
44	■	×	×	×	×	■	55.91%	0.07%
45	■	×	×	×	■	×	50.80%	9.15%
46	■	■	×	×	■	×	50.79%	0.01%
47	■	■	■	×	■	×	50.78%	0.03%
48	■	×	■	×	■	×	50.78%	0.00%
49	■	■	×	■	×	×	49.72%	2.09%
50	■	×	■	■	×	×	49.69%	0.05%
51	■	×	×	■	×	×	49.63%	0.12%
52	■	■	■	■	×	×	49.61%	0.05%
53	■	■	■	■	■	■	47.85%	3.55%
54	■	×	■	■	■	■	47.84%	0.01%
55	■	■	×	■	■	■	47.82%	0.03%
56	■	×	×	■	■	■	47.82%	0.01%
57	■	×	×	■	×	■	47.12%	1.45%
58	■	■	■	■	×	■	47.11%	0.04%
59	■	×	■	■	×	■	47.10%	0.01%
60	■	■	×	■	×	■	47.07%	0.06%
61	■	■	■	■	■	×	42.72%	9.24%
62	■	■	×	■	■	×	42.72%	0.01%
63	■	×	■	■	■	×	42.72%	0.01%
64	■	×	×	■	■	×	42.67%	0.11%

Results for Machine 6

Test Id	2D Texture Mapping	Alpha Blending	Depth Testing	Stencil Testing	Dithering	Alpha Testing	Percentage of Peak Fill	Percentage Change
1	×	×	×	×	×	■	100.00%	
2	×	×	■	×	×	■	99.48%	0.52%
3	×	×	×	×	×	×	98.71%	0.77%
4	×	×	×	×	■	×	97.11%	1.63%
5	×	×	■	×	■	■	96.00%	1.14%
6	×	×	■	×	■	×	95.60%	0.42%
7	×	×	■	×	×	×	95.55%	0.05%
8	×	×	×	×	■	■	94.88%	0.70%
9	×	■	×	■	■	×	61.16%	35.54%
10	×	×	■	■	×	×	60.80%	0.59%
11	×	×	■	■	■	×	60.18%	1.02%
12	×	×	×	■	■	■	60.06%	0.19%
13	×	■	■	×	■	■	59.98%	0.14%
14	×	■	×	■	×	■	59.92%	0.09%
15	×	■	■	×	■	×	59.87%	0.09%
16	×	■	■	×	×	■	59.81%	0.09%
17	×	■	■	■	■	■	59.71%	0.17%
18	×	×	■	■	■	■	59.60%	0.19%
19	×	■	×	×	×	■	59.48%	0.19%
20	×	■	■	■	■	×	59.44%	0.08%
21	×	■	■	×	×	×	59.38%	0.09%
22	×	×	■	■	×	■	59.25%	0.23%
23	×	■	■	■	×	■	59.10%	0.25%
24	×	■	×	■	■	■	59.09%	0.03%
25	×	■	×	×	■	×	59.09%	0.00%
26	×	■	×	×	×	×	59.06%	0.05%
27	×	■	×	×	■	■	58.91%	0.25%
28	×	■	■	■	×	×	58.90%	0.01%
29	×	×	×	■	×	×	58.87%	0.06%
30	×	×	×	■	×	■	58.66%	0.34%
31	×	■	×	■	×	×	58.56%	0.17%
32	×	×	×	■	■	×	57.29%	2.17%
33	■	×	×	×	■	■	40.11%	29.99%
34	■	■	■	■	×	×	40.08%	0.09%
35	■	■	×	×	×	×	40.02%	0.14%
36	■	×	■	■	×	■	39.97%	0.12%
37	■	×	■	■	■	■	39.95%	0.05%
38	■	■	■	×	×	■	39.74%	0.54%
39	■	×	×	■	×	×	39.72%	0.04%
40	■	■	■	■	×	■	39.66%	0.14%
41	■	■	■	■	■	×	39.64%	0.05%
42	■	■	■	×	■	×	39.62%	0.06%
43	■	■	×	■	■	■	39.56%	0.14%
44	■	×	■	■	■	×	39.56%	0.01%
45	■	×	■	■	×	×	39.46%	0.26%
46	■	×	×	×	×	■	39.43%	0.06%
47	■	■	■	×	■	■	39.39%	0.11%
48	■	■	×	×	×	■	39.37%	0.06%
49	■	×	■	×	×	■	39.28%	0.23%
50	■	■	■	×	×	×	39.25%	0.07%
51	■	×	×	■	■	■	39.25%	0.01%
52	■	×	×	■	×	■	39.23%	0.06%
53	■	×	×	■	■	×	39.21%	0.03%
54	■	■	×	×	■	×	39.12%	0.25%
55	■	×	×	×	■	×	39.07%	0.12%
56	■	×	■	×	■	×	39.05%	0.04%
57	■	×	×	×	×	×	39.00%	0.14%
58	■	■	■	■	■	■	38.85%	0.38%
59	■	■	×	■	×	×	38.81%	0.09%
60	■	■	×	×	■	■	38.63%	0.46%
61	■	×	■	×	■	■	38.63%	0.01%
62	■	■	×	■	■	×	38.58%	0.13%
63	■	■	×	■	×	■	38.28%	0.77%
64	■	×	■	×	×	×	38.27%	0.03%

Results for Machine 7

Test Id	2D Texture Mapping	Alpha Blending	Depth Testing	Stencil Testing	Dithering	Alpha Testing	Percentage of Peak Fill	Percentage Change
1	×	■	■	×	×	×	100.00%	
2	×	×	■	×	×	×	99.88%	0.12%
3	×	×	×	×	×	×	99.17%	0.71%
4	×	■	×	×	×	×	99.13%	0.04%
5	■	■	■	×	×	×	84.64%	14.62%
6	■	×	■	×	×	×	84.53%	0.13%
7	■	×	×	×	×	×	84.03%	0.59%
8	■	■	×	×	×	×	84.02%	0.01%
9	×	×	×	■	×	×	73.02%	13.09%
10	×	■	■	■	×	×	72.97%	0.06%
11	×	×	■	■	×	×	72.96%	0.02%
12	×	■	×	■	×	×	72.91%	0.06%
13	■	×	×	■	×	×	59.87%	17.89%
14	■	■	×	■	×	×	59.86%	0.02%
15	■	■	■	■	×	×	59.84%	0.04%
16	■	×	■	■	×	×	59.76%	0.13%
17	×	×	×	×	■	×	54.85%	8.22%
18	×	×	×	×	×	■	54.84%	0.03%
19	×	■	×	×	■	■	54.82%	0.02%
20	×	■	■	×	■	×	54.80%	0.04%
21	×	×	■	×	■	×	54.80%	0.01%
22	×	×	■	×	■	■	54.80%	0.00%
23	×	×	×	×	■	■	54.79%	0.02%
24	×	■	×	×	■	×	54.78%	0.01%
25	×	■	×	×	×	■	54.77%	0.01%
26	×	■	■	×	×	■	54.74%	0.05%
27	×	×	■	×	×	■	54.72%	0.04%
28	×	■	■	×	■	■	54.70%	0.05%
29	■	■	×	×	■	×	47.28%	13.56%
30	■	×	×	×	■	×	47.25%	0.05%
31	■	×	×	×	■	■	47.24%	0.03%
32	■	×	×	×	×	■	47.22%	0.03%
33	■	■	×	×	×	■	47.21%	0.03%
34	■	■	×	×	■	■	47.21%	0.01%
35	■	×	■	×	■	×	47.17%	0.08%
36	■	■	■	×	×	■	47.10%	0.14%
37	■	×	■	×	×	■	47.09%	0.02%
38	■	■	■	×	■	■	47.09%	0.01%
39	■	×	■	×	■	■	47.09%	0.01%
40	■	■	■	×	■	×	47.08%	0.02%
41	×	×	■	■	■	×	43.47%	7.67%
42	×	■	×	■	×	■	43.44%	0.07%
43	×	×	×	■	■	×	43.42%	0.03%
44	×	■	×	■	■	×	43.41%	0.04%
45	×	■	■	■	■	■	43.40%	0.02%
46	×	×	×	■	×	■	43.40%	0.01%
47	×	×	■	■	■	■	43.38%	0.03%
48	×	■	■	■	■	×	43.38%	0.01%
49	×	×	■	■	×	■	43.36%	0.03%
50	×	■	■	■	×	■	43.36%	0.01%
51	×	■	×	■	■	■	43.31%	0.10%
52	×	×	×	■	■	■	43.30%	0.03%
53	■	■	×	■	■	×	38.07%	12.09%
54	■	■	×	■	■	■	38.06%	0.02%
55	■	×	×	■	■	■	38.05%	0.02%
56	■	×	×	■	■	×	38.04%	0.03%
57	■	×	■	■	■	×	38.01%	0.10%
58	■	×	■	■	×	■	38.01%	0.00%
59	■	×	■	■	■	■	38.00%	0.01%
60	■	■	■	■	■	×	37.99%	0.02%
61	■	■	■	■	■	■	37.99%	0.00%
62	■	■	×	■	×	■	37.96%	0.08%
63	■	■	■	■	×	■	37.96%	0.02%
64	■	×	×	■	×	■	37.75%	0.54%

Results for Machine 8

Test Id	2D Texture Mapping	Alpha Blending	Depth Testing	Stencil Testing	Dithering	Alpha Testing	Percentage of Peak Fill	Percentage Change
1	■	■	■	×	■	×	100.00%	
2	×	■	×	×	×	×	27.09%	72.91%
3	×	×	■	×	×	×	27.05%	0.14%
4	×	×	×	×	×	×	27.04%	0.03%
5	×	■	■	×	■	■	27.01%	0.11%
6	×	■	■	×	■	×	26.99%	0.08%
7	×	■	×	×	■	×	26.99%	0.01%
8	×	×	■	×	■	×	26.96%	0.10%
9	×	×	×	×	■	×	26.93%	0.09%
10	×	■	■	×	×	×	26.75%	0.68%
11	×	×	■	×	■	■	22.71%	15.11%
12	×	×	×	■	×	■	19.27%	15.16%
13	■	×	■	×	■	×	19.17%	0.52%
14	■	×	×	×	×	×	19.16%	0.06%
15	×	×	×	■	■	■	19.15%	0.02%
16	■	■	×	×	×	×	19.15%	0.02%
17	■	×	×	×	■	×	19.14%	0.07%
18	■	■	■	×	×	×	19.12%	0.08%
19	■	×	■	×	×	×	19.11%	0.06%
20	×	■	×	■	×	■	19.10%	0.03%
21	×	×	■	■	×	■	19.06%	0.22%
22	×	■	■	■	×	■	19.05%	0.05%
23	×	■	■	■	■	■	19.05%	0.01%
24	×	■	×	■	■	■	19.03%	0.09%
25	■	■	×	×	■	×	18.99%	0.23%
26	×	×	■	■	■	■	18.88%	0.60%
27	×	×	■	■	■	×	18.19%	3.63%
28	×	×	×	■	×	×	18.10%	0.51%
29	×	■	■	■	■	×	18.09%	0.02%
30	×	×	×	■	■	×	18.06%	0.21%
31	×	■	×	■	×	×	17.93%	0.69%
32	×	×	■	■	×	×	16.96%	5.43%
33	×	■	×	■	■	×	16.62%	1.97%
34	■	■	■	■	■	×	14.94%	10.10%
35	■	■	×	■	■	■	14.80%	0.94%
36	■	×	■	■	■	■	14.78%	0.18%
37	■	×	×	■	■	■	14.78%	0.00%
38	■	×	×	■	×	■	14.74%	0.28%
39	■	×	■	■	×	■	14.73%	0.02%
40	■	■	■	■	■	■	14.73%	0.01%
41	■	■	■	■	×	■	14.70%	0.21%
42	■	■	×	■	×	■	14.63%	0.50%
43	■	■	■	■	×	×	14.54%	0.56%
44	■	■	×	■	×	×	14.40%	0.98%
45	■	×	■	×	■	■	14.37%	0.20%
46	■	×	×	■	×	×	14.15%	1.54%
47	■	■	■	×	■	■	13.56%	4.15%
48	■	■	×	■	■	×	11.66%	14.06%
49	■	×	■	■	×	×	10.34%	11.26%
50	■	×	■	■	■	×	9.79%	5.34%
51	■	×	×	■	■	×	7.76%	20.74%
52	×	■	■	■	×	×	7.42%	4.41%
53	×	×	×	×	■	■	1.83%	75.35%
54	×	■	×	×	×	■	1.82%	0.39%
55	×	×	×	×	×	■	1.82%	0.26%
56	×	■	×	×	■	■	1.81%	0.59%
57	×	×	■	×	×	■	1.41%	22.10%
58	×	■	■	×	×	■	1.41%	0.01%
59	■	■	×	×	×	■	0.05%	96.41%
60	■	×	×	×	×	■	0.05%	0.77%
61	■	×	■	×	×	■	0.05%	0.15%
62	■	■	■	×	×	■	0.05%	0.37%
63	■	×	×	×	■	■	0.05%	0.33%
64	■	■	×	×	■	■	0.03%	30.88%

Results for Machine 9

Test Id	2D Texture Mapping	Alpha Blending	Depth Testing	Stencil Testing	Dithering	Alpha Testing	Percentage of Peak Fill	Percentage Change
1	×	×	×	×	×	×	100.00%	
2	×	×	■	×	×	×	99.90%	0.10%
3	×	■	×	×	×	×	99.86%	0.04%
4	×	■	■	×	×	×	99.81%	0.05%
5	■	■	×	×	×	×	97.66%	2.15%
6	■	×	■	×	×	×	97.64%	0.02%
7	■	■	■	×	×	×	97.59%	0.05%
8	■	×	×	×	×	×	97.49%	0.10%
9	×	■	■	×	■	×	86.42%	11.35%
10	×	×	■	×	■	×	86.40%	0.03%
11	×	■	×	×	■	×	86.39%	0.01%
12	×	×	×	×	■	×	86.37%	0.03%
13	×	■	■	×	×	■	85.55%	0.94%
14	×	■	×	×	×	■	85.53%	0.03%
15	×	×	■	×	×	■	84.68%	1.00%
16	×	×	■	×	■	■	83.64%	1.22%
17	×	×	×	×	■	■	83.29%	0.42%
18	×	■	×	×	■	■	83.04%	0.30%
19	×	■	■	×	■	■	83.04%	0.01%
20	×	×	×	×	×	■	80.26%	3.34%
21	■	×	×	×	■	×	79.75%	0.64%
22	■	×	■	×	■	×	79.68%	0.08%
23	■	■	×	×	■	×	79.67%	0.02%
24	■	×	×	×	×	■	79.31%	0.45%
25	■	■	■	×	■	×	79.31%	0.00%
26	■	×	■	×	×	■	79.21%	0.13%
27	■	■	■	×	■	■	79.19%	0.02%
28	■	×	■	×	■	■	78.34%	1.07%
29	■	■	■	×	×	■	78.03%	0.39%
30	■	■	×	×	■	■	77.40%	0.81%
31	■	■	×	×	×	■	71.80%	7.23%
32	■	×	×	×	■	■	71.64%	0.22%
33	×	■	×	■	×	×	64.59%	9.84%
34	×	×	■	■	×	×	64.59%	0.00%
35	×	■	■	■	×	×	64.58%	0.02%
36	×	×	×	■	×	×	64.55%	0.04%
37	×	×	■	■	×	■	61.21%	5.17%
38	×	■	×	■	■	■	61.19%	0.03%
39	×	×	×	■	■	×	61.18%	0.01%
40	×	×	×	■	■	■	61.12%	0.10%
41	×	×	×	■	×	■	61.12%	0.01%
42	×	■	×	■	×	■	61.11%	0.00%
43	×	×	■	■	■	■	61.11%	0.01%
44	×	■	×	■	■	×	61.10%	0.01%
45	×	■	■	■	×	■	61.09%	0.02%
46	×	■	■	■	■	■	61.09%	0.01%
47	×	×	■	■	■	×	61.07%	0.02%
48	×	■	■	■	■	×	60.87%	0.34%
49	■	×	×	■	×	×	54.16%	11.02%
50	■	■	■	■	×	×	54.13%	0.05%
51	■	■	×	■	×	×	54.12%	0.03%
52	■	×	■	■	×	×	53.89%	0.42%
53	■	×	■	■	■	×	51.91%	3.66%
54	■	×	×	■	■	■	51.91%	0.01%
55	■	■	×	■	■	■	51.88%	0.05%
56	■	■	×	■	×	■	51.87%	0.04%
57	■	■	■	■	■	×	51.86%	0.00%
58	■	×	×	■	×	■	51.84%	0.04%
59	■	■	■	■	■	■	51.84%	0.00%
60	■	×	■	■	×	■	51.84%	0.00%
61	■	×	■	■	■	■	51.84%	0.00%
62	■	■	×	■	■	×	51.82%	0.03%
63	■	×	×	■	■	×	51.82%	0.01%
64	■	■	■	■	×	■	51.77%	0.09%

Results for Machine 10