SIGGRAPH 2001
Course # 3

Performance OpenGL: Platform Independent Techniques

or

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

Dave Shreiner
shreiner@sgi.com

July 18, 2001

Abstract

Given that the OpenGL application programming interface (API) is available on a multitude of different computing platforms, how can one structure their code for maximum portability and best performance? This paper attempts to discuss some of the underlying commonalities of OpenGL implementations, how simple considerations kept in mind while authoring an application can greatly help in achieving better performance across all OpenGL capable platforms, and present techniques to determine where the performance bottleneck (and there always will be one) is occurring.

Introduction

OpenGL is the most widely accepted cross-platform API for rendering computer generated images. 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 makes it also well suited for the experienced developer to develop real-time, interactive applications. As with any low-level programming mechanism (e.g. assembly languages for microprocessors), it's just as easy to develop fast, efficient OpenGL programs as it is to write a perfectly valid OpenGL command sequence, and suffer considerably less than optimal performance.

This paper hopes to illustrate some of the conditions that can hamper OpenGL performance regardless of which platform the application is executing on, and to further present strategies to help determine which methods for rendering pixels and geometry are optimal for a particular platform without exploring platform specific extensions.

Many of the techniques discussed in this paper can be explored in the OpenGL Sample Implementation (SI) [1]. 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 representatives that I 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 best combinations of techniques that will provide the best performance for that platform.

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

Errors in OpenGL can have disastrous affects on application performance. As OpenGL doesn't actively alert you to when an error occurs, and does its best to make sure that your application continues uninterrupted, many developer's never bother to check to see if their application is generating any OpenGL errors.

As mandated by the OpenGL specification, no input into OpenGL 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 are made) after the internal OpenGL error state is set to represent the situation that occurred. Once one error has occurred, the OpenGL error state is not updated until that error has been cleared (by checking for an error).

Determining if you have any errors

Checking for OpenGL errors is simple. A call to `glGetError()` will return the code for one of OpenGL’s seven errors. As an implementation may track more than one error (this in not in contradiction to the above; 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 in such a case).

A simple C-preprocessor macro can be very helpful in detecting errors for individual calls, 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"`).

This macro can be placed almost anywhere in an application, except between a `glBegin()`/`glEnd()` sequence. In such cases, utilize the macro after the `glEnd()` call.

Just as continuing development on a program that crashes may not be the most prudent approach, allowing errors in your OpenGL application can have drastic effects on your performance.

Errors with Feedback and Selection Mode

Selection and feedback modes provide error information slightly differently than OpenGL state setting commands. As both of these render modes require a buffer to write their return data into, if their results overflow the buffer provided, they don’t specify an error detectable with `glGetError()`, but rather return a zero (which is returned when the next `glRenderMode()` is issued), to indicate that an error has occurred.

Determine which part of the OpenGL pipeline is performance limiting factor

Regardless of how well written an application is, it will always have one section that executes slower than all of the other sections. It 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.” Having a bottleneck, and every program always has one, is completely different than having a program that isn’t executing up to one’s performance expectations. However, if the program’s executing slower than is required, then the first place to look for performance improvements is where the bottleneck occurs.

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*.

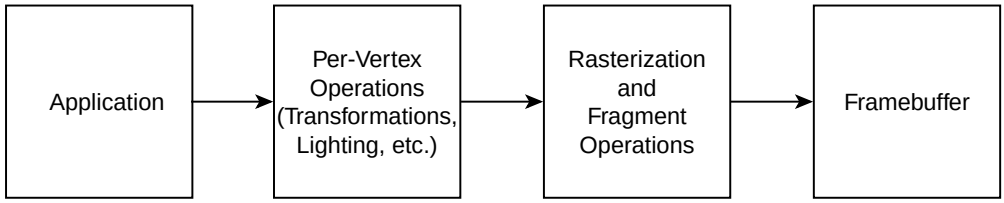

Figure 1: Block diagram of OpenGL pipeline phases

The transformation phase is responsible for vertex transformations, lighting, and other operations, and produce OpenGL *fragments* to be passed into the rasterization phase.

Rasterization is responsible for coloring the appropriate pixels in the framebuffer. This phase includes operations like depth testing, texture mapping, and alpha blending, among others.

"Bottleneck" determination

With an understanding of how the OpenGL pipeline operates, it becomes much easier to understand which section of your application or OpenGL 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 meet the performance requirements set upon it. 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 metric), then one of the above situations will always exist.

Techniques for determining pixel-fill limitations

Fill-limited applications, as mentioned, are not able to color all the pixels in the viewport in the alloted time. This situation results from too much being required for some set (possible all) of pixels.

Of the three performance situations, pixel-fill limitations is perhaps the simplest to determine. 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.

This immediately presents two possible solutions: reduce the size of the viewport for the applications execution, or do less per-pixel processing.

Techniques for determining transformation limitations

If reducing the size of the viewport doesn't increase the frame-rate of the application, then the application isn't fill limited. In such a case, a test can be conducted to determine if the bottleneck is in the transformation section of the rendering pipeline, or in the application.

To determine if the application is vertex limited, change 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])
#define glVertex2fv(v)        glNormal3f(v[0],v[1])
#define glVertex2iv(v)        glNormal3i(v[0],v[1])
#define glVertex2sv(v)        glNormal3s(v[0],v[1])
#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 transformation, with most of the real work being done with the subsequent `glBitmap()` or `glDrawPixels()` calls, which have little impact on transformation bottlenecks.

The reason that this technique is useful is that it transfers the 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 operation is limited to a few variable assignments).

To generate a useful performance metric, measure and record the duration for 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 not fill or 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 re-analyze the choices made for storing data that is to be passed to OpenGL, and employ a code profiling tool (such as the `pixie` and `prof` tools available in Unix) to attempt to make the application perform more efficiently.

Performance Optimizations for each stage of the OpenGL pipeline

Techniques for reducing per-pixel operations

Expanding the rasterization section of the OpenGL pipeline, we see that there are 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 understanding what the application does, and what type of visual quality is required, you can have a measure of control over the application's performance.

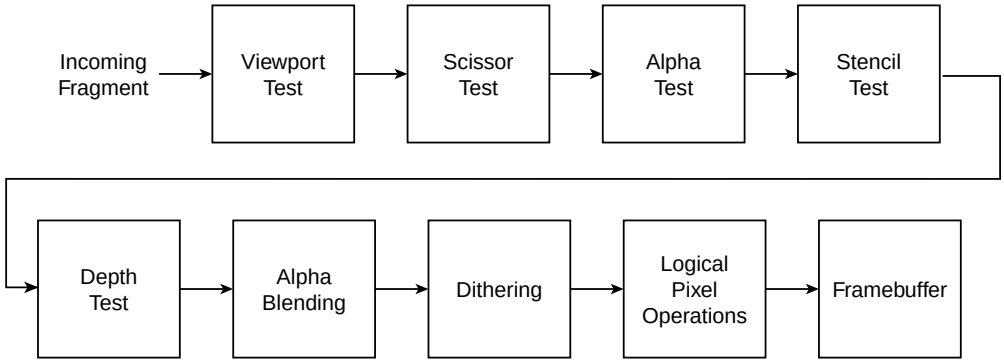

Figure 2: Stages of fragment processing pipeline

The operations in the rasterization pipeline can be classified into two categories: color-application 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 testing operations include depth buffering, stencil tests, alpha-value tests, and viewport and scissor box tests.

Reducing the visual quality of the rendered image may be 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:

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 loose considerable color resolution, which may result in banding in the image. Dithering can help to reduce the banding, but also adds some additional fragment processing.

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 as such under the appropriate conditions, do not need to rendered. This technique is really a geometric transformation technique (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.). This may not be a limiting factor depending upon the scene's visual complexity; just like shadows, the side effects of backface culling may not be noticeable in an interactive application.

Utilize a lesser quality texture mapping minification filter

OpenGL supports four 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_MIPMAP_NEAREST`, which only requires a single texel lookup per fragment. The tradeoff, once again, is visual quality. However, when considerations are made for the screen-space size of the object are made, 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 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 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.

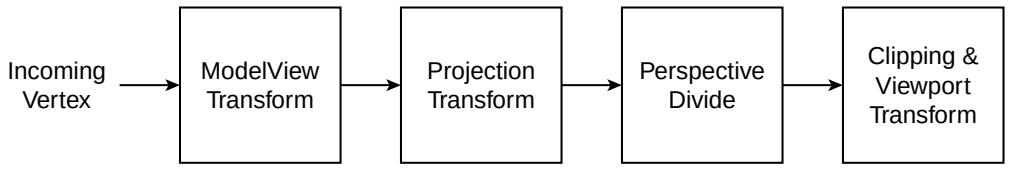

Figure 3: Stages of transformation pipeline

Be cognizant of the work done at a vertex

The amount of computation done for a vertex can vary greatly depending upon which modes are enabled. Every vertex is transformed by the model-view and projection matrices, as well as being clip-tested. If lighting, texture coordinate generation, or user-defined clipping planes is enabled, then the amount of work for a single vertex increases significantly. Of the modes that can be enabled for a vertex, lighting is usually the most heavily used and costly, but also the most flexible in its usage options.

Lighting, along with other performance considerations for transform limited applications are discussed in the next sections.

Use connected primitives . . . or don't

As previously mentioned, every vertex is transformed, clipped, and perspective divided on its way to becoming a fragment. For many geometric 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. To that end, 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. 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.

`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:

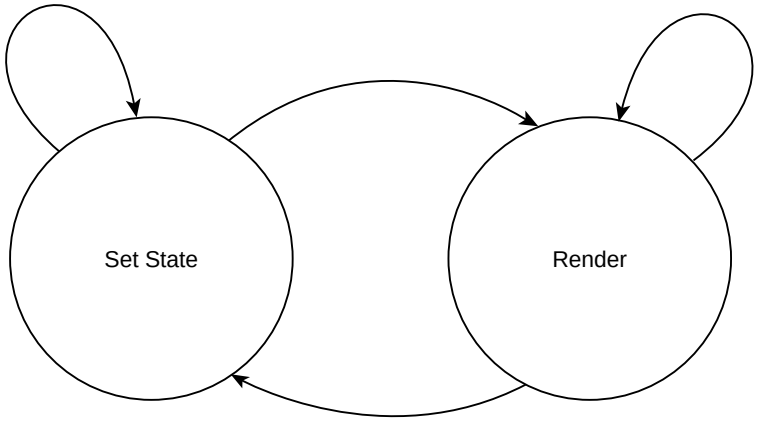

Figure 4: An application programmer's view of the OpenGL state machine

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 4 would look like:

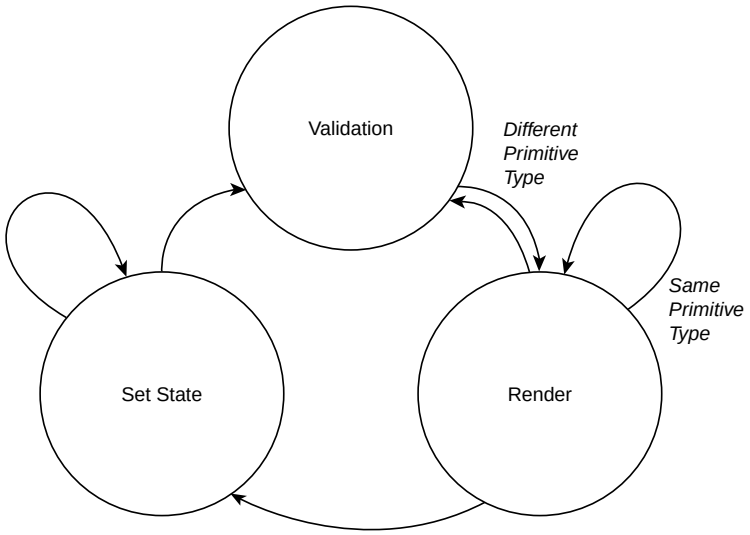

Figure 5: OpenGL state machine taking into consideration the validation phase

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 conducted, and the parts of the OpenGL machine that were affected by the state changes are updated, in this case, 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. For something a little different, 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 3

```
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 results. The values have been normalized.

Code Technique	Machine 1	Machine 2	Machine 3
Example 4	3.99	4.36	2.23
Example 5	3.99	4.50	2.15
Example 6	1.00	1.00	1.00

Table 2: Normalized results for various computing platforms

For each of the architectures, Example 6 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 3), 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 per platform.

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 help 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.

Determining the best way to pass geometry to the pipe

OpenGL has four ways of transmitting geometric data to the OpenGL pipeline: immediate mode, display lists, vertex arrays, and interleaved arrays. Although all four 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 developer's are most accustomed to. The major disadvantage to immediate mode, particularly for specifying geometry, is that a large number of function calls are required. On some computing architectures where function call overhead can have an impact, immediate mode may not be the fastest mode of execution.However, it may also be the case that the other modes of rendering utilize immediate mode calls in their internals.

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 later at display list execution time. Optimization of the token stream may occur, but is not prevalent in most drivers. In such a case, display list performance may not offer any increase over immediate mode performance.

However, if the graphics hardware has dedicated display list memory or processing capabilities, this mode may 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 enter into the execution performance of vertex arrays.

Finally, interleaved arrays are very similar to vertex arrays, however, as compared to vertex arrays, the data for each vertex is stored in a contiguous block of memory. This method lends itself very nicely to storing vertex data into a structure. Once again, for certain architectures, this may be the best format for specifying geometry. The only way to know is to conduct a benchmark case.

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 done due to floating-point comparison issues.

General OpenGL Performance Techniques

State Sorting - Trying to organize rendering around state changes

As previously mentioned, every time some state values (excluding vertex data such as color, normals, etc.) are updated, OpenGL prepares to do a validation process at the next rendering operation. Given that premise, a logical optimization step is 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.

Obviously, as objects can have considerably different states, and determining a precedence for which states to sort for first is recommended. Generally, the best approach 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 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

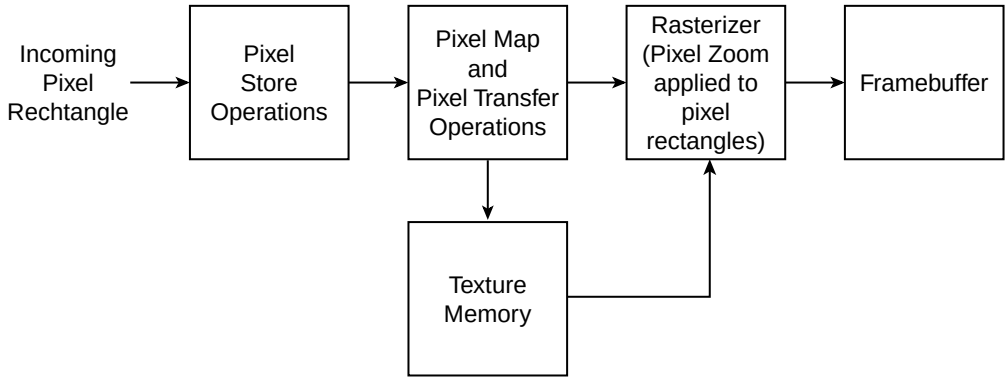

Figure 6: OpenGL's pixel conversion and transfer pipeline

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. Depending upon the resolution, format, and number of components, this conversion could take a while to complete. Minimizing these conversion is a good approach. 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 supported formats for any hardware available today; they should be avoided in all but the simplest cases.

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 saving.

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 transfered 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, but it's unlikely this optimization is present in may 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.

Conclusion

This short tutorial hopes to provide OpenGL programmers with some good habits such that any OpenGL application they write for any platform will have a solid basis for performance tuning. The habits discussed are not rocket science, but handy tips that every OpenGL programmer should tuck into their tool box. One benefit of what's been presented is the fact that there are no platform dependencies (e.g. extensions) introduced; these ideas work across various graphics hardware and operating systems.

One final point to be reiterated: to get the maximum performance from your OpenGL application, benchmark and verify that all of the assumptions for all of the hardware platforms that you intend to deploy your application on are true. Many situations have arisen where application programmers have assumed that since one idiom of OpenGL programming has worked on one platform, it must work the same on all platforms. A little investigation at the start of a project will be returned many times over when optimization is applied in earnest.

Glossary

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.

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.

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.

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 may be updated occasionally. The latest copy of the notes can be found at

`http://www.shreiner.net/SIGGRAPH/s2001/Performance.OpenGL/`

References

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

[2] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics Comptuer Systems Inc., 1.2.1 edition, 1999.

[3] The OpenGL Architecture Review Board and Dave Shreiner, editors. *The OpenGL Reference Manual*. Addison-Wesley, 3rd Edition edition, 1999.

[4] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *The OpenGL Programming Guide*. Addison-Wesley, 3rd Edition edition, 1999.