Optimizing OpenGL Data Throughput on Mac OS X

Apple has built a highly optimized and fully modern implementation of OpenGL into the very heart of Mac OS X. This article explains general OpenGL optimization techniques and specifically how to optimize your OpenGL code to maximize vertex data throughput. It is intended for developers with an OpenGL application that needs optimizing, a basic understanding of OpenGL programming, and a good idea of how their application's data is structured and used.

Introduction

Efforts to improve the efficiency of OpenGL applications usually involve several concepts: finding ways to minimize OpenGL workloads, balancing processing load among the GPU and one or more CPUs, and minimizing function calls that cause synchronization between the CPU and GPU. You will also want to make sure that your code follows the most efficient and best optimized routes through OpenGL, the "fast paths" in the system. To the extent that your OpenGL code does not adhere to these tenets, you are wasting CPU processing time and system power and you are leaving extra performance on the table.

In order to squeeze the highest possible performance out of your OpenGL applications, you'll need to know where the OpenGL fast paths are and how to avoid the speed bumps along the way. OpenGL has two general types of data, vertex and texture. Vertex data defines information used in vertex processing while texture, or image, data defines inputs to fragment processing. In this article, you'll learn the general concepts of OpenGL optimization as well as the best ways to handle static and dynamic vertex data on Mac OS X. This article will also discuss the pros and cons of the various OpenGL drawing methods, including immediate mode, display lists, vertex arrays (vertex array ranges and vertex array objects), and vertex buffer objects.

In order to get the most out of this article, you'll need a basic understanding of how OpenGL works and the characteristics of your application data. If you need an OpenGL refresher course, or if you are new to OpenGL on the Mac platform, see the ADC OpenGL page for more information.

General Concepts for OpenGL Optimization

This section talks about optimization concepts in more detail since they apply in general to all OpenGL implementations and embody ideas underlying many of the optimization guidelines discussed in subsequent sections.

Use `GLfloat`s to Minimize CPU Conversions

Mac OS X normally uses `GLfloat` for its internal processing. By providing your data in `GLfloat` format, you minimize the number of conversions and copies that the OpenGL driver must perform before sending your data on to the video card.

Minimize CPU Copies

In its default mode, OpenGL will make several copies of your vertex data. First, immediate mode requires that OpenGL make a copy of the current vertex data as each vertex is submitted. This article assumes that you'll skip the current vertex copy by avoiding immediate mode in favor of display lists, vertex arrays, or vertex buffer objects. Second, OpenGL must also make a copy of any vertex data for the command buffer that is sent to the video card. Methods for avoiding this second copy differ for static and dynamic data so there are two later sections that explain optimization strategies for static and dynamic data respectively.

Minimize Function Call Overhead and Command Buffer Traffic

In graphics intensive applications, even the miniscule amount of time spent calling an OpenGL function can have a significant effect on performance if your code makes too many function calls. Thankfully, the OpenGL implementation shipped with Mac OS X provides several ways to reduce this overhead. First, in order to avoid the expense of calling a function for every single bit of vertex data, avoid using immediate mode if at all possible. By using "bulk" APIs like display lists or vertex arrays, you can amortize the cost of the function call across many vertices. Second, for those function calls that you can't eliminate, you can maintain your own current context and use the macros in `CGLMacro.h` to remove one level of function call overhead.

Minimize State Changes

OpenGL is a state machine and all state changes have some amount of overhead. In some cases this overhead will show up at the

next drawing command rather than at the time of the state change itself, but the end result is that all state changes in OpenGL have the potential to be expensive to some degree and can heavily impact the performance of your application. In order to minimize state changes in the OpenGL state machine, you should group your drawing operations according to state. For example, draw all of your lit primitives first before drawing your unlit primitives to avoid having to turn lights on and off multiple times per frame, which can be very expensive. Additionally, developers should avoid thrashing state by issuing multiple changes to the same piece of state between drawing commands. In the end, these multiple changes are completely redundant and just waste CPU time.

Maximize Asynchronous Behavior Between CPU and GPU
The CPU and GPU are both powerful processors which, with proper care, can be used completely asynchronously. In OpenGL, there are several obvious and other not-so-obvious ways to cause either the CPU or GPU to stall on the other. By eliminating these synchronization points in your code, you'll free up the processor and graphics card to do what they do best, process your data. First, never call `glFinish`. There is no reason to call `glFinish` and calling it will only force your application to stall while OpenGL waits for everything in the command pipeline to complete on the graphics card. Second, you should minimize your use of the closely related `glFlush` API. Typically, the only time you'll want to call `glFlush` is when the CPU and GPU are referencing the same data. This scenario can arise when you are using ARB Vertex Buffer Objects, Apple Vertex Array Range, or Apple Texture Range with a DYNAMIC (SHARED for the Apple extensions) storage type. In this case, you'll want to isolate the CPU and GPU from one another by double buffering your data and only calling `glFlush` when you switch which buffer you are writing to. The `glFlush` should be called as soon as a buffer has been filled and is ready for submission to the GPU. However, you must be careful to properly set up the double buffering so that you don't issue too many `glFlush` calls, which can cause the system to run out of OpenGL command buffers and force the CPU to go synchronous with the GPU.

Use Threads
The Mac OS X OpenGL implementation is not thread-safe but you can still split your processing onto multiple threads. It is extremely important for developers not to issue OpenGL commands for a single context from multiple threads without proper thread synchronization. This is a leading cause of both rendering anomalies and application crashes. There are three specific ways to use threads with OpenGL. First, use one thread per context, which will not require any special thread synchronization. Second, use standard thread synchronization techniques with a single context to ensure that multiple threads do not enter OpenGL simultaneously. Please note that the OpenGL Profiler has a thread safety check (turn on "Break on thread error" in the Breakpoints window) that can be used to help ensure this requirement is maintained. Finally, in a technique that is particularly relevant for this article, you can split the computation of vertex data off onto one thread while another thread handles the drawing of that data. You can gain performance advantages by applying threads on single processor machines but threads really shine on multiple CPU machines since each processor can devote itself to a thread, thereby potentially doubling your throughput.

Drawing with Vertex Data

Immediate mode is the simplest and most straightforward of the drawing APIs in OpenGL. Unfortunately, immediate mode doesn't scale well so you'll need to switch over to one of the other APIs for better performance as your data set grows. This section will give you a brief look at your API choices and describe the pros and cons of each one. The goal of these more advanced APIs is to allow you to bundle up your geometry and data and hand it to OpenGL in a single function call. In the case of static data, you can also have OpenGL cache your data in VRAM and then make a single call to draw it whenever you wish.

Display List
Display lists are a mechanism for encapsulating drawing commands (both immediate mode and array commands) and data, designed to allow optimized "create once, use many" scenarios. The difference between normal drawing commands and a display list is that once you create a display list, the contents are immutable and thus can be optimized by OpenGL. This can allow for huge performance improvements which are in line with some of the techniques discussed below. Because the contents of a display list are static, OpenGL can optimize the contents and cache the result on the video card. Due to the fact that you can't change the contents of a display list once you've created it, display lists lend themselves better to static data than to dynamic data. The more often you draw with a display list, the more performance benefit you will realize. Drawing a display list is as simple as calling `glCallList` with the list name.

Vertex Array Range
Vertex arrays allow you to assemble your vertex coordinates, colors, normals, and other vertex data into large arrays. To use vertex arrays you give OpenGL the pertinent information, like which arrays you want to use, where they are, how big they are, etc. Then, you make a simple call to draw all or a subset of the vertices in the arrays.

Vertex Buffer Object
Vertex buffer objects are similar to vertex arrays in that they encapsulate blocks of vertex data into easily manageable groups. There are some important differences between VBOs and VARs, however. First, VBOs have the advantage of allowing you to pull data from multiple VBOs when you draw. This allows you to split your data into multiple chunks to better fit the nature of the data. For example, you could have one VBO hold your static vertex position data while another VBO held your dynamic vertex color data. The static VBO could be cached in VRAM while the dynamic VBO remained in main memory, freeing you from flushing the static data across the bus every time the dynamic data changed. Second, while VARs give you explicit control over what data gets flushed and when, with VBOs flushing is done implicitly and you may get more flushes than necessary. This can be particularly problematic when you have streaming (dynamic) data and need to flush very specific ranges of vertices.

Vertex Array Object
A vertex array object is an encapsulation of vertex array state (usable with both vertex array range and vertex buffer object) that you can use to more efficiently set up the current vertex array state for drawing. Simply create a VAO, bind it, and set up the state the way you want it using vertex array ranges or vertex buffer objects. When you're ready to draw with that state later on, simply

bind the VAO again and start drawing. The pseudo code looks like this:

```
        //      create and bind your VAO
        glGenVertexArraysAPPLE(1, &vertexArrayObjectID);
        glBindVertexArrayAPPLE(vertexArrayObjectID);

        //      set up your vertex state for this VAO

        //      clear out the vertex state
        glBindVertexArrayAPPLE(0);

        ...more code here

        //      when you're ready to draw, bind the VAO again
        glBindVertexArrayAPPLE(vertexArrayObjectID);

        //      do your array drawing - here we use glDrawArrays
        glDrawArrays(drawMode, 0, triCount);
```

Handling Static Vertex Data

Your first priority in optimizing static vertex data throughput is to get off the immediate mode path. While Apple is working hard to optimize all parts of OpenGL, immediate mode has inherently too much overhead for even moderate data sets. Generally, on Mac OS X developers should strive to reduce the number of function calls in performance critical code, which will obviously reduce overall function call overhead. Immediate mode requires a number of function calls for every vertex drawn, thus incurring a high overhead. Additionally, in immediate mode, OpenGL must copy every single set of vertex data into the current vertex. If you've got static vertex data, you can make use of display lists, vertex arrays, or vertex buffer objects to dramatically improve your performance. In the case of display lists, the code changes required are minimal and the performance rewards can be quite impressive.

Using Display Lists

The code to create and use display lists is quite simple:

```
        //      creating a display list
        GLuint listname = glGenLists( 1 );
        glNewList( listname, GL_COMPILE );

                //      normal immediate mode drawing here
                //      use glBegin/glEnd as required
                //      you may also use DrawArrays, etc.

        glEndList();

        //      drawing with a display list
        glCallList( listname );

        //      disposing of a display list
        glDeleteLists( listname, 1 );
```

Note: There are some things to keep in mind when using display lists. First, you'll get the best performance when using more than 16 vertices per list. Second, in order to make the driver's job easier, provide consistent data for each vertex in a list. For example, provide normal, color, and vertex data for each vertex rather than leaving out color or normal for some vertices. Third, it is very important that you provide all attributes that may be dynamic within a single draw command between `glBegin` and the first `glVertex`. For example: `glBegin`, `glVertex`, `glColor`, `glVertex` cannot be optimized but `glBegin`, `glColor`, `glVertex`, `glColor`, `glVertex` can be optimized. Fourth, as with all OpenGL drawing, it is very desirable to have as few state changes as possible within a display list. If you can group together your `GL_TRIANGLES`, `GL_QUADS`, `GL_POINTS`, and `GL_LINES` of similar mode and state, the display list optimizer may be able to batch them together for more efficient drawing. Finally, decomposing strips and fans into triangles and quads can also allow for more behind the scenes display list optimization. Keep in mind, though, that such decomposition will increase the number of vertices that the GPU must transform, potentially creating a new bottleneck.

Using Vertex Arrays

Like display lists, vertex arrays allow you to skip the current vertex and command buffer copies of your vertex data. However, you

may need to reorganize your data in order to use vertex arrays. See the ADC OpenGL and OpenGL Standard web sites for more information on the basics of vertex arrays. For static vertices, you'll want to tell the OpenGL driver to cache the vertex array data in VRAM in order to achieve the best performance. On Mac OS X, you ask the driver to cache the vertex data in VRAM by calling `glVertexArrayParameteriApple` with the `GL_STORAGE_CACHED_APPLE` hint as shown below. Once you've defined your vertex data and set the proper storage hint, you can draw vertices from the array using `glDrawArrays`, `glDrawElements`, or `glDrawRangeElements`. If you need to synchronize against a vertex array, you can use the `APPLE_fence extension` to check if OpenGL is done with a particular set of commands.

Here is the setup for a vertex array that will be cached in VRAM:

```
        //      standard vertex array setup
        glEnableClientState( GL_VERTEX_ARRAY);
        glVertexPointer( 3, GL_FLOAT, stride, vertex_ptr );

        //      tell OpenGL to cache the vertex data in VRAM
        glVertexArrayParameteriAPPLE(
                GL_VERTEX_ARRAY_STORAGE_HINT_APPLE,
                GL_STORAGE_CACHED_APPLE );

        //      Map the vertex data into AGP space
        glVertexArrayRangeApple( array_size, vertex_ptr );

        //      mark the data as changed so that OpenGL
        //      knows that it needs to be uploaded to the card
        glFlushVertexArrayRangeAPPLE( array_size, vertex_ptr );
```

Note: The main advantage of vertex arrays is that they can handle huge amounts of data much more easily than display lists. In many cases, your application's vertex data may already be in arrays suitable for use with the vertex array APIs. If that's the case and the data is static, you can easily switch over to caching your vertex data in VRAM for a potentially large performance boost.

Using Vertex Buffer Objects

Vertex buffer objects are similar to vertex array ranges in many ways but there are several important differences. First, unlike VARs where you provide your own data and simply tell OpenGL to access it, with VBOs you let OpenGL manage the buffers for you. You tell OpenGL how much space you need, ask OpenGL for an appropriately sized buffer, fill or modify it as necessary, and then release the buffer back to OpenGL before you can draw with it. Second, while VARs require that all the data be static or dynamic and you can only draw using one VAR at a time, VBOs let you mix and match buffers. For example, you could store your static vertex positions in one VBO that is cached in VRAM, place your dynamic vertex colors in a second VBO, and draw your geometry using both VBOs at once. Finally, unlike VARs where you handle flushing explicitly and can precisely control the range of elements that are flushed, with VBOs OpenGL handles the flushing implicitly. Also note that while OpenGL is required to flush the entire vertex buffer object, VBOs give you the ability to divide your data up somewhat since you can draw using multiple VBOs at once.

The code to set up a vertex buffer object for static data looks like this:

```
        //      generate the buffer ID and bind it
        glGenBuffersARB( 1, &vertexBufferID );
        glBindBufferARB( GL_ARRAY_BUFFER_ARB, vertexBufferID );

        //      tell OpenGL how much data you've got
        //      and whether it is static (GL_STATIC_DRAW_ARB),
        //      dynamic (GL_DYNAMIC_DRAW_ARB),
        //      or streamed (GL_STREAM_DRAW_ARB)
        //      according to the ARB spec, streamed means
        //      "specify once and use once or perhaps only a few times"
        glBufferDataARB( GL_ARRAY_BUFFER_ARB, dataSize,
                NULL, GL_STATIC_DRAW_ARB );

        //      ask OpenGL for a buffer to store your data
        GLubyte *dest = glMapBufferARB( GL_ARRAY_BUFFER_ARB, GL_WRITE_ONLY_ARB );

        //      write your data into the buffer once

        //      flush the buffer object and let OpenGL cache it
        glUnmapBufferARB( GL_ARRAY_BUFFER_ARB );
```

```
//      unbind the VBO
glBindBufferARB( GL_ARRAY_BUFFER_ARB, 0 );
```

Handling Dynamic Vertex Data

As with static data, your first priority should be to get off the immediate mode path. Since display lists aren't suitable for dynamic data, you'll need to switch over to vertex array ranges, vertex array objects, or vertex buffer objects.

Using Vertex Arrays

Vertex arrays can be applied to dynamic vertex data in much the same way as static vertex data. There are really only three differences. First, you must use a different storage hint to tell OpenGL to use the copy of the vertex data in your application's address space rather than cache the data in VRAM. Second, because OpenGL will be DMAing the vertex data directly from your application's copy of the data, it becomes the responsibility of your application to keep the vertex data around until OpenGL is finished drawing with that data. You can use an `APPLE_fence` for OpenGL synchronization to make sure you know when OpenGL is done with your data. Note, fences should not be used for thread synchronization. For that, use standard thread synchronization primitives instead. Third, since both your application and OpenGL will need to access the vertex data, you'll get better performance if you double buffer your dynamic vertex data. Modify one frame's worth of data while OpenGL renders from the previous frame's data. In this way you minimize the synchronization overhead and neither the CPU nor the GPU will need to block against the other waiting for access to the vertex data. As with the static vertex data, you can draw with dynamic vertex data using `glDrawArrays`, `glDrawElements`, and `glDrawRangeElements`. The dynamic vertex array setup code looks like this:

```
//      standard vertex array setup
glEnableClientState( GL_VERTEX_ARRAY);
glVertexPointer( 3, GL_FLOAT, stride, vertex_ptr );

//      tell OpenGL to cache the vertex data in VRAM
glVertexArrayParameteriAPPLE( GL_VERTEX_ARRAY_STORAGE_HINT_APPLE, GL_STORAGE_SHARED_APPLE );

//      Map the vertex data into AGP space
glVertexArrayRangeApple( array_size, vertex_ptr );

//      mark the data as changed so that OpenGL knows that it needs to be uploaded to the card
glFlushVertexArrayRangeAPPLE( array_size, vertex_ptr );
```

Using Vertex Buffer Objects

Using vertex buffer objects with dynamic data is the same as using them with static data except for the constant passed into `glBufferDataARB`. For dynamic data, you need to use `GL_DYNAMIC_DRAW_ARB` instead of `GL_STATIC_DRAW_ARB`.

Mixed Data

There's one last important topic to cover. What if your data isn't completely static or dynamic? Perhaps you've got static vertex positions but your vertex colors or texture coordinates vary from frame to frame. In this case, you'll want to take advantage of the flexibility that vertex buffer objects offer to draw using more than one VBO at once. Use one static VBO to cache your vertex positions and a second dynamic VBO to hold your changing colors.

Conclusions

This article has covered a variety of optimization choices but it is important to remember the underlying optimization goals: don't do any work you don't have to do, take maximum advantage of the available CPU and GPU hardware, and let your hardware resources work as independently as possible. Choose the optimization path that best fits your data and that allows you to load balance the work between the GPU and CPU. Cache static data in VRAM. Double buffer dynamic data and only update it when absolutely necessary. Finally, use the OpenGL Profiler, OpenGL Driver Monitor and CHUD tools early and often to fully understand your application's performance characteristics.

For More Information

- See the ADC OpenGL Topic Page for more information and resources on OpenGL on Mac OS X.
- See the OpenGL.org Home Page for cross-platform information.
- See the ADC Performance Topic Page for more information and resources on optimizing performance on Mac OS X.
- See the ADC Tools Topic Page for more information on Mac OS X optimization tools, including the CHUD suite of tools.
- See the article OpenGL Tools for Serious Graphics Development for a description of the OpenGL tools available on Mac OS X and how to use them to analyze and optimize graphics.

Posted: 2005-03-21

Get information on Apple products.
Visit the Apple Store online or at retail locations.
1-800-MY-APPLE