Programming OpenGL for modern hardware

(Originally by Eskil Steenberg, eskil@obsession.se, edited by Stefan Gustavson, stegu@itn.liu.se)

This is a short guide to programming modern, large OpenGL applications, beyond simple tutorials. Emphasis is put on performance, maintainability and portability.

Hardware vs. software performance

When OpenGL was originally created, graphics hardware was slow and the CPU was usually not the bottleneck. Back then you could use immediate mode (`glBegin, glEnd`) for serious work. Today, however, the graphics cards run circles around CPUs, so the main performance problems can be in the API and in the driver software. For each command you give the GL driver, it has to check various state, check for errors, and make state changes to the hardware. All this takes considerable time, so try to make as few GL function calls as possible. This is done by drawing large objects rather than small, and to reduce the number of state changes by sorting the things you want to draw based on their their use of textures, lights, shaders and such. Drawing 10 polygons in a vertex array isn't much slower than drawing only one, it's the draw call that costs. You need to send serious amounts of work to OpenGL for the hardware in a modern GPU to become the bottleneck. (In Direct3D, the draw call overhead is actually huge. DX10 is better than DX9, but it still has a much larger call overhead than OpenGL.)

In modern use of OpenGL, you need to look out for API call overhead. Using immediate mode, or using strategies like drawing each particle in a particle system as an individual object, is *very* bad.

What to use

OpenGL is a large and old API. There are often several ways of doing the same thing, but some of them are there only for backwards compatibility and should be avoided. So how do you choose what to use? It's good to build your own subset of functions which you need and then stick to that. Often GL programs have problems with state leaks, and to a large degree that can be avoided by not messing with every possible state that can be found in GL. To select what parts of GL to use, there are three things to consider: *functionality*, *performance*, and *support* (i.e. if the features are supported, or if there are acceptable fallbacks if they are not). A good place to start reading is the extension registry at `www.opengl.org`. ATI and NVIDIA also have extensive documentation, but they tend to be biased towards their own extensions. Usually ”ARB” extensions are the best. They are mature and widely supported, and are usually very well designed. When you are considering using an extension, check its date. If it is older than a few years, the same functionality may have entered the core OpenGL API, or there might be an newer and better extension for the same purpose. The fact that an extension is formally supported by a driver doesn't mean that its necessarily good to use. It may just be there for older applications. It is generally wise to only use extensions that are supported by multiple vendors and keep your code as portable as possible, but sometimes you want to make an exception and code for the feature set of current generation GPUs from a specific vendor.

Here is a list of useful, widely supported and recommended extensions. (Note that some of these depend on other extensions, which are also used.) Your own requirements may be different.

- Frame buffer objects (FBO)
- Stencil Depth interleaved
- Vertex buffer objects (VBO)
- Float buffer
- Stencil two-sided
- Shading language 100
- Cube mapping
- Frame buffer blit
- Half_float (for texture formats)
- S3 Texture compression

Geometry

Getting geometry into the graphics card memory is a large bottleneck. The thing to avoid more than anything else is immediate mode: *never* make a call to `glVertex`. Even when you have unpredictable data like particle systems or implicit surfaces where the number of vertices change dynamically, it is better to build up a buffer in software and then draw it as a vertex array. The reason is not only the call overhead, but also that the user pattern of immediate mode is so unpredictable that it is almost impossible to optimize. If you really need functionality where you add one vertex at a time, build a function that fills a software buffer. When you have filled that buffer, draw the buffer using a vertex array and then start over. The code structure will be similar to immediate mode, but a lot faster.

In short, *there is no reason at all to use immediate mode* for any serious amount of work. (It's OK if you store your immediate mode calls in a display list, but then you are not actually using immediate mode, you are using the OpenGL driver to build a much more efficient data structure. See below.)

So what should you use? There are three different good ways to send geometry data to OpenGL: *vertex arrays*, *vertex array buffers* and *display lists*.

Vertex arrays are simple to use and supported by all OpenGL implementations. When using them, try to use one array for all vertex data (position, normal, color, texcoord etc) and to interleave the data. This way the graphics card will be able to fetch the data more efficiently. For maximum performance, make the data as small as possible by using small data types (`float` rather than `double`, `short` rather than `int`) and by not sending any unused data.

Vertex array objects let you map a part of the graphics cards memory and then use that memory for your vertex arrays. This is supported by almost all hardware and is easy to implement if you already have vertex array support. One thing to remember here is that reading and writing to this memory can be a lot slower than reading and writing to main memory, so avoid using it in tight loops like this:

```
for(i = 0; i <  1000; i++)
      membuf[x] += weight[i];
```

Instead, use a local variable and update once:

```
for(i = 0; i <  1000; i++)
      f += weight[i];
membuf[x] = f;
```

In other words: compute first, and when you are done, put it into the vertex array. Vertex array buffers are OpenGL state, and it can be expensive to switch between them. It may be a good idea to use fewer larger buffers and then store the geometry of many objects in the same buffer, rather than having one buffer for each small object.

Display lists are among the oldest methods of speeding up OpenGL drawing, and it is still a fast and useful way of sending geometry data to the GPU. Unfortunately, it is not very flexible. You use a display list to record a long sequence of OpenGL calls, even immediate mode calls if you like, and then you can execute that sequence again and again. The driver can optimize a display list significantly, because it knows exactly what the user wants done. Display lists are useful for non-animated background set pieces that need to be drawn over and over. A thing to note is that a display list can be drawn with any shader, so you can actually animate objects placed in a display list by means of a vertex shader. When you record a display list, record as much state as possible in the list to reduce state change overhead. And whatever you do, if you record a stream of immediate mode calls, do not forget to include the `glBegin` and `glEnd` calls! When you compile a display list, don't draw the display list while you are recording it - it is usually faster to record the list first and then display it.

When using vertex arrays it is always good to have a reference array and reference the same vertex

multiple times rather than duplicating vertex data. Failing that, GPU hardware usually has a vertex cache, so a vertex used twice may only need to be computed once, but the vertex cache is limited in size. If you want the vertex cache to work for you, draw neighboring polygons after each other. Drawing the polygons in a object in a random order is not recommended.

Shaders

All modern GPUs have shader-capable hardware. Even if you only use the old fixed functionality transform and lighting, the driver will actually use a built-in shader that performs these functions for you. There is no actual fixed function hardware any more, and therefore it makes sense to use the shader functionality. The best way to create shaders in OpenGL is to use GLSL - OpenGL Shading Language. Contrary to Direct3Ds HLSL, the compiler for GLSL sits in the driver, so the application can actually generate shaders on the fly and send them to the driver for execution. It generally makes sense for larger applications to have a system that can generate the needed shaders dynamically.

When writing shaders, some people tend to write ”uber-shaders”, large shaders that encompass all different things the application wants to do. This, however, will make the shaders slow, so it's better to have many different, more specialized and smaller shaders. It can be good to have one shader for lighting involving only one light source, and then a very similar shader with two light sources, rather than one general shader with a light count parameter. If a shader has an `if` statement, it may actually end up computing *both* branches because of constraints from the parallell execution, so be careful not to introduce dead code. Remember that each instruction gets multiplied with the number of pixels you are drawing with the shader active, so a small change can make a big difference.

Generally speaking, graphics hardware is orders of magnitude faster than a CPU, so it makes sense to use the GPU for as much as possible. Programmable hardware can be used for other things than pure 3D rendering, like tessellation, animation, compositing and fluid simulation.

Render surface

Avoid rendering to the front buffer, P-buffers and accumulation buffers. If you need to render to a texture or an image, use *frame buffer objects (FBOs)*. If you need to copy from one image or texture to another, use the ”frame buffer blit” extension. Current ATI hardware does not support FBOs with stencil buffers, so if you need to render to a texture and use stencil buffer on ATI today, render to the framebuffer and blit into the texture. If you are using FBOs on NVIDIA hardware or future ATI hardware and drivers and need stencil buffers, always use interleaved stencil/depth buffer as this is the only supported format.

Texture formats

Try to avoid exotic texture formats. Use GL_RGB and GL_RGBA. For massive texture amounts, S3 texture compression is supported in hardware and saves memory also on the GPU. For all HDRI textures use GL_FLOAT_RGB and GL_FLOAT_RGBA. On NVIDIA hardware GL_HALF_RGB and GL_HALF_RGBA are also available. They use a 16 bit floating point type which is notably faster and uses less memory. Conversion from 32-bit ”float” to 16-bit ”half” can be slow and is best performed in advance, not at texture load time. Cube, 1D, 2D and shadow mapping is always available, but 3D texturing (although often available) is often slow. Avoid hyperbolic texturing.

General tips

- Make as few calls as possible. It's far better to draw one large object than many small ones.
- Change of GL state takes time, especially switching programs, buffers, vertex arrays and FBOs. Avoid state changes where possible, and state sort where possible.
- The graphics card bus is usually a big bottleneck, so avoid sending textures or geometry data between the main memory and the GL driver for every frame if you can avoid it. This can be a huge cost! Use the OpenGL texture functions to keep textures resident, and use vertex buffer objects to keep geometry data in GPU memory.
- Try to wrap GL calls and keep them together so that they can be switched out when new functionality gets in to GL. Try to have as few OpenGL entry points in the code as possible.
- Don't use feedback mode, other than possibly for debugging purposes. It's all in software nowadays, so it's better to have your own implementation for such things.
- Try to draw the non-transparent parts of the scene from front to back. Modern hardware checks Z-buffering before it computes the fragment shader, so if you draw the closest surfaces first the shaders on the surfaces behind them don't need to run.
- In very shader-heavy scenes it may be a good idea to draw a pure Z-pass first, so that no shader is computed when not needed.
- Back face culling is essentially free, so use it!
- Smooth lines can usually not be drawn with shaders.
- Always make your shaders use as little memory as possible. Even if it runs fine on your machine, it may run out of memory on another system and generate an error or switch to software mode. Sometimes shader compilers use resources like texture samplers without telling the user, so try not to allocate more resources than you need.
- Where possible, move computation from the fragment shader to the vertex shader.
- "noise" is a function in GLSL that doesn't work on current ATI and NVIDIA hardware. Use a texture look-up instead, or write your own noise function in GLSL.
- If you are unsure, contact developer relations of NVIDIA, ATI, Intel and Apple (Apple writes their own drivers for both NVIDIA, ATI and Intel hardware, so they are not the same as on PC)
- Always profile, both on the CPU side and on the GPU side. You need to know where your application spends its time before you can speed it up.
- Test on multiple systems, with different operating system and hardware vendors.
- 3D textures can be very slow. Current GPUs have texture units optimized for 2D texturing.
- Loops with a dynamic loop count are allowed in modern GLSL, but using a fixed loop count can be considerably faster.
- Use debuggers like GLDebug to make your GLSL debugging easier. Vendors give out free licenses to non-commercial developers.
- If you have a large OpenGL project, ask your hardware vendors to profile the application for you. They can give you great tips on how to improve it, and they have an interest in getting your software to run well on their hardware even if it is a non-commercial project.

Most recently updated by Stefan Gustavson (stegu@itn.liu.se) 2009-03-18