Good Shader Practices

A CSCI 1230 Guide

Contents

1 Introduction

This guide is focused on explaining a few good mechanical practices for making your GPU happy. This includes efficient data specification and code optimizations. This guide does not explain the costs and benefits of higher level designs, such as forward vs deferred shading.

2 Geometry

2.1 Indexed Buffers

For all but the simplest of geometry, vertices are used more than once in a given mesh. A single vertex is in turn made of a position vector, as well as possibly a normal vector, multiple texture coordinates, and so forth. It is therefore very expensive memory-wise to keep redundant vertices in your arrays. Indexing arrays is a solution to this problem: List every vertex once in one array, and then specify another array whose entries are indices into the first array. The entries in this index array specify which vertices are to be used to form each primitive.

An index array may look like $\{(0, 1, 2), (0, 2, 3), (2, 3, 4) \ldots\}$. Vertices 0, 1, and 2 will form the first triangle, vertices 0, 2, and 3 will form the second, and so forth.

2.2 Interleaved Buffers

The major unit of computation for the vertex shader is vertices. Not positions, or normals, or texture coordinates - each excecution of the shader requires that all information regarding a single *vertex* be retrieved from memory. For this reason, it is handy to organize our memory so that all of the vertex data is in one location in memory. Consider two examples:

1. You specify three VBOs - one for position data, one for normal data, and one for texture coordinate data. Visually, the buffers look like:

   [PPP, PPP, PPP ...] , [NNN, NNN, NNN ...] , [TT, TT, TT ...].

   This is not spatially optimal. The GPU must read from three different locations in memory in order to gather all the knowledge for a vertex.

2. You specify one VBO - the position, normal, and texture data for each vertex are listed as a single group. Visually, the buffer looks like:

   [PPPNNNTT, PPPNNNTT, PPPNNNTT, ...]

   This is far better. Now when the position data is read from memory, the normal and texture memory is as close as possible. This is a good cache optimization scheme.

The second example is called vertex interleaving. For N vertex attributes, you've now saved yourself N-1 memory lookups. Hurray!

2.3 Indexed Interleaved Buffers

In care you're wondering, you can indeed (and should where possible) combine indexing and interleaving. You will reap the benefits of your memory thrift as you afford more geometry and effects drawn than before.

2.4 Degenerate Triangle Strips

Note: This is pretty clever.

The most straightforward way to draw triangles is with `GL_TRIANGLES` — every three elements constitutes a distinct triangle, wholly independent of the last. However, `GL_TRIANGLE_STRIP` is a much faster way of accomplishing the same task. Organizing your geometry buffer such that you draw in strips saves you space in your index buffer *and* improves memory access speed, as two of the vertices in the previous triangle are always present in the next.

But wait, drawing with strips means you'll have lots of draw calls - one for each strip. That will slow everything down, right? And just appending each strip in the same call will cause them all to be connected by incorrect triangle artifacts. What to do?

Degenerate triangles to the rescue! You can pack everything into a single draw call by placing "caps" at the ends of your triangle strips. This is done by specifying the last vertex of the first strip and the first vertex of the second strip twice. Visually, your index buffer (for a width of 50) should look like:

[... 48, 98, 49, 99, **99**, **50**, 50, 100, 51, 101, ...]

Instead of having incorrectly shoved your strips together, you've specified four triangles of zero area between them as a safe connection. Because they have zero area, they are never rendered. As a further benefit, the winding order of the triangles remains the same, so you don't have to worry about reversing every other strip. Basically, they do what you want them to do.

2.5 Material groups

In large enough scenes, there will be many meshes and many materials with which to render them. In order to save on GPU calls (such as switching uniforms or even shaders), it is helpful to group your geometry by material. Set your shader and uniforms, draw everything with those settings, and move onto the next group. This will require pre-sorting your objects somehow.

Depending on the scope of your project, you may never notice the benefits of material grouping, especially in this class. You should investigate other more impactful optimizations before you spend time on material grouping, and make sure that implementing material groups doesn't cost more than it saves!

3 GLSL Optimizations

3.1 Default Optimizations

The GLSL compiler optimizes shaders nearly to a fault. While optimizations are vendor-specific, there are a number of assorted optimizations that will commonly be done for you. Knowing them will let you more meaningfully concentrate your efforts and can also explain changes in performance if you switch hardware.

- **Unnecessary variables and functions are removed.** If the flow of the shader's logic is such that a certain variable is never used or a function is never reached, it is completely removed from the shader. This can prove troubling when trying to debug a shader, as you may get OpenGL-side errors about missing variables (e.g. when you try to set them from C++).

  One hack to prevent this is to multiply the uniform or function's return value by an epsilon value (e.g. 0.00001) and add it value that's always used (perhaps at the end of your main). This way your shader will still "use" the values, thus they will not get optimized out, and your final output will not be meaningfully altered.

- **Uniform computations are *sometimes* precomputed when possible.** If you multiply uniform A by uniform B in an equation, the compiler might recognize that this too is a uniform (as a function of strictly uniforms is itself *uniform*), so will not execute the operation for every piece of input data (e.g. vertex or fragment). However, you should not rely on this and should precompute such expressions beforehand.

3.2 Fast Language Features

GLSL is built specifically for GPUs, which are in turn built specifically for doing common graphics operations as fast as possible. It is helpful to know about these features when coding your shaders. You will not only be spared from reinventing the wheel, but also know how to accomplish your tasks more efficiently. Some operations on normal CPUs have certain performance prices, while their prices on GPUs are much lower.

3.2.1 Swizzle

GLSL allows you to address the components of vectors in an arbitrary order. Consider the vector vec4 foo. The components of foo can be "swizzled" like so:

```
foo.xyz = foo.wzy
```

This switches the y and z components of the vector and replaced x with w. You can switch any components around, and you don't have to use all the components at once (notice how foo is a vec4 but we only assigned 3 components). You can also repeat components (e.g. foo.xy = foo.zz). Swizzling is a good practice for making your shader faster and more readable over constructing vectors on a per-element basis (such as `vec3 foo = vec3(bar.y, bar.x, bar.z)`).

3.2.2 MAD Operations

MAD - Multiply, then Add. MAD operations are very low cost, so writing your arithmetic as

```
float a = 0.5 * b + 0.5 ;
```

rather than

```
float a = 0.5 * ( 1.0 + b );
```

will indeed be faster. While the compiler *might* be able to recognize this toy example despite the ordering (especially considering the constants), structuring your algebra to take advantage of this can save you time. For example, you may find that factoring/premultiplying/grouping your terms in such a way will leave you with the same algebraic result but with fewer instructions.

3.2.3 `Step` function

The GLSL `step` function checks whether its second argument is less than its first, returning 0 if true and 1 otherwise. It works on a component-wise basis. That is to say, if you compare two `vec4`s, the result will be a `vec4` whose components are the result of the component-by-component comparison of the arguments.

The `step` function is useful because you can use two at once to effectively create a ternary operator. Branching on GPUs is not recommended, so if you can turn small branches into `step` functions, do it. See section 3.3 for more.

3.2.4 `Mix` function

GLSL's mix function is a linear interpolation function, or lerp for short. GPUs are good at lerping, so avail yourself of the GLSL mix function wherever you can.

`mix( a, b, t)` lerps between a and b based on the value in t. The last argument can be a float to lerp between all components of the vector uniformly, or it can be the same type as `a` and `b` to lerp on a component-wise basis.

3.2.5 `Smoothstep` function

`Smoothstep` does a Hermite interpolation between two endpoints based on the value of the third element. The name is misleading - `smoothstep` looks like a curved version of `mix` rather than a binary function like `step`. You can use smoothstep if you want a slightly fancier version of `mix`.

3.3 Avoiding Branches

GPUs are great at a lot of things, but sadly they are not so great at evaluating conditional branches. Branches are a great way to slow down GPU code. However, there are a few techniques for avoiding them.

- **Don't have branches.** Lame suggestion, right? Perhaps not. If your shader program is accounting for multiple possible outcomes and checking for various settings, you will most likely increase your performance by using separate shaders for separate pieces of functionality. For example, rather than checking for valid inputs in GLSL, only pass valid inputs from the C++ side. The only reason using multiple shaders wouldn't help is if each shader draws so little that the cost of constantly switching through them overtakes the cost of the conditional branch.
- **Use the `step` function.** While simply removing branching may work for complete pieces of functionality, some pieces of functionality may themselves *need* conditionals in order to be evaluated properly - just consider some long logical mathematical formula that calculates later results differently based on outcomes of earlier results. The `step` function is a branchless conditional - it returns either 0 or 1 depending on the arguments you give it.

  Consider the following logic:

```
if( conditional==true )
{
```

```
        answer += foo;
}
else
{
        answer += bar;
}
```

You can stick that conditoinal into a `step` function to get the following:

```
float cond = step( conditional );
answer += cond*foo + (1.0-cond)*bar;
```

Regardless of what the value of the conditional is, you should be able to turn it into an inequality fairly easily.

4 Additional Resources

This list of suggestions is far from exhaustive. The following is a short list

4.1 OpenGL Wiki

The Wiki can be far more helpful when learning about how to use OpenGL functions, and why things are set up the way they are. You'll often find practical examples and explanations of the arguments to functions, critical clarification often missing from the docs.

4.2 OpenGL 4.0 Shading Language Cookbook

Now on its second edition, the Cookbook is a fantastic resource for learning how to do shaders. It takes you through all of GLSL's language features while also explaining how to implement various effects such as bloom and shadow mapping. Every argument of every function is explained, as is each step of each algorithm, so you'll feel confident in writing shaders by the time you've read it.

4.3 Shadertoy

Shadertoy is an incredible website showcasing thousands of shader demos. What's more, you can see all of the source code right next to the shader. You can also change it, recompile it, and see what happens. This is best place to learn advanced fragment shading techniques such as fractal generation, noise generation, and raymarching. You can also see how people use the functions mentioned in section `3.2` - pictures are worth a thousand words. It's run by Pixar's own Iñigo Quelez, so you may learn a thing or two.

4.4 Further Reading

- GPU Pro 1, 2, 3, 4, and 5
- GPU Gems 1, 2, and 3
- Physically Based Rendering, Second Edition: From Theory to Implementation