15-462 Project 3 Checkpoint: Introduction to OpenGL Shader Programming

Release Date: Tuesday, February 21, 2012

Checkpoint Due: Thursday, March 08, 2012, 23:59:59
(No late days!)

Starter Code: http://www.cs.cmu.edu/afs/cs/academic/class/15462-f11/www/project/p3.tar.gz

Useful references:
GLSL Quick Reference Guide: http://www.cs.cmu.edu/afs/cs/academic/class/15462-s10/www/proj/glslref.pdf
GLSL Full Language Specification: http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf
OpenGL Reference Pages: http://www.opengl.org/sdk/docs/man/
GLSL Tutorials:
http://www.lighthouse3d.com/opengl/glsl/
http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/
GLSL Lecture:
http://www.cs.cmu.edu/afs/cs/academic/class/15462-s12/www/lec_slides/lec11.pdf

1 Introduction

This document describes the requirements needed for the checkpoint. The checkpoint aims to introduce you to basic GLSL programming. The second part of the project will focus more on taking advantages of shaders to create interesting effects in rendering models.

2 Checkpoint requirements

Here is what you need to do for the checkpoint:

1. Read/skim through the 5 resources at the begining of this document to get high level idea about shaders.
2. Have a look at the example shader we provide to see how they work. You can then just remove the provided shader, or modify it to become your own shader.

3. Remove the shader and render the scene using fixed functionality (we already gave you this, i.e. you do not need to do any camera or object transformation)
4. Render the color buffer and the depth buffer to textures.
5. Write a simple shader to render the color buffer texture in a second pass without modifying it. So, you'll get the exact same output as if rendered without any shaders.
6. Submit the project as usual to:
   `/afs/cs.cmu.edu/academic/class/15462-f11-users/`*andrewid*
   `/p3-checkpoint/`.
   Note that you DO NOT need to submit screenshots or do any writeup.txt for this checkpoint. You just need to submit all the necessary files, i.e. src files, model files, shader files etc so that we can compile and run your program successfully.

3 Implementation details

3.1 Scene Format

Rather than making you render the scene, we simply provide you with a function that renders the scene for you. Your algorithm should work independently of what is drawn in the scene, so you should not need to know the details of the rendering function.

There is one exception: it necessary to know the near and far planes used when rendering to convert depth values correctly. Therefore, we provide access to the camera. However, you **do not** need to set the transformation matrices using the camera. This is done for you. You only use the camera to access values needed for your shaders.

Therefore, you won't actually be sending any primitives to the GPU, as the starter code does that for you. You should set up your shader code and use the function we give you to render the scene, collecting the data you need to create outlines. You can can (will almost definitely need to) call it multiple times.

For convenience, all scenes will be approximately the same size and contain similarly-size objects, so you should not have to tweak the shaders to get decent results for different scenes. The function we give you will not modify the OpenGL state; that is, the OpenGL state will be the same after the call as before.

3.2 More on Shaders

Here's a bit more detail on shader programming, though you should use other resources for a more comprehensive description. While programming, the quick-ref card is an invaluable resource. This is more just some random details that are important for this lab.

3.2.1 Debugging Shaders

Unfortunately, there are not many good, free solutions for debugging shaders, particularly on the school machines. You don't even have print statements to help you. Therefore, it will take some patience and creativity to figure out what is wrong with your program. Instead of print statements, you may have to rely on a lot of test code to help pinpoint which lines of your shader are not working correctly.[1]

3.2.2 Variable Modifiers

While local variables are pretty much the same as C, global variables in a shader have some modifiers that affect how they are used and created. You may recall that in fixed-functionality, there were two types of variables you can send to the GPU, uniform and attribute. Both these kinds can be used in shaders, along with a few others:

const These are constants that must be set where declared and cannot be changed. They are identical in both the vertex and fragment shaders.

uniform These are set by the application and cannot be modified by the shaders. You declare them in the shader and set them in the C++ program using the `glUniform` family of functions. They must be the same for an entire primitive. Most of the OpenGL state with which which you are familiar (e.g., the model-view matrix, material colors, light position) are actually built-in uniform variables which you do not need declare. The quick-ref card lists these. Uniforms can be read by both vertex and fragment shaders.

attribute These are per-vertex data. Like uniforms, the shader declares them and they are set by the application. Vertex, normal, and texture coordinate are examples of built-in attributes, which are listed on the reference card. Attributes can only be read by the vertex shader (since they are per-vertex data). You can also create custom attributes, though you shouldn't need any for this lab.

varying These are used to pass intermediate data from the vertex shader to the fragment shader. They must be set by the vertex shader. The values are linearly interpolated for each fragment using each vertex in the primitive and then passed into the fragment shader, where they are read-only.

Note: Be very careful with `varying` variables. Since they are linearly interpolated, they are not suitable to pass certain data. For example, passing a

[1] One possible trick for fragment shaders is to output and intermediatevariable as the final color, which will give you a visualization of that variablefor each pixel. You'll of course need to come up with some way of representing your variable as a color.

transformation matrix will almost certainly not work, since the linear combination of transformation matrices won't necessarily result in a transformation similar to the orignal ones.

3.3 OpenGL Extensions

Since newer OpenGL functions aren't supported on all graphics cards, most systems don't actually have header files with all the OpenGL functions in them. In fact, nearly every function added to OpenGL in the past 10 years in not available by default on most systems. OpenGL uses an extension mechanism to gain access to newer functions.

We take care of most of the process for you using an external library call GLEW. The only thing of which you need to be aware is that most functions in OpenGL are appended with letters indicating that they are an extension. For example, nearly all functions associated with shaders are appended with `ARB`, e.g., `glUniform1fARB` instead of `glUniform1f`. Also, not all graphics cards will support these features, so your machine may not have them available. The course staff highly suggests you make sure your video driver is up-to-date to ensure that you have access to all OpenGL functions of which your card is capable.

3.4 Using Textures in Shaders

As always, consult the OpenGL documentation for more detailed information about this.

One of the easiest methods for doing an image-based computation with a shader (e.g. outlines) is using OpenGL textures. You put the image data into textures. Then, you render a single, screen-sized quad, setting the texture coordinates such that the texture coordinate at each pixel corresponds to the matching pixel of the texture. For example:

```
... // rendering passes that store data in textures

... // bind textures and set shader

... // set projection and modelview to identity

// this assumes you're using GL_TEXTURE_RECTANGLE_ARB
glBegin( GL_QUADS );
glTexCoord2f( 0.0f, 0.0f );
glVertex3f( -1.0f, -1.0f, -1.0f );
glTexCoord2f( width, 0.0f );
glVertex3f(  1.0f, -1.0f, -1.0f );
glTexCoord2f( width, height );
glVertex3f(  1.0f,  1.0f, -1.0f );
glTexCoord2f( 0.0f, height );
```

```
glVertex3f( -1.0f,  1.0f, -1.0f );
glEnd();
```

You set the vertex shader to pass on the texture coordinate, which will be interpolated for the fragment shader. Then you can use the fragment shader to do per-pixel computation, looking at any neighboring pixels you wish.

3.4.1 Texture Rectangle

While using normal 2D textures is possible, there are limitations. For most GPUs, 2D texture dimensions must be a power of 2,[2] which doesn't exactly match the screen size. So OpenGL has a texture rectangle extension, which lets you create textures of any size, say w pixels in width and h pixels in height. The texture coordinates then run from $(0, 0)$ to (w, h) rather than $(0, 0)$ to $(1, 1)$, which makes it more natural and precise to lookup exact pixels. We suggest you use `GL_TEXTURE_RECTANGLE` rather than `GL_TEXTURE_2D`.[3]

3.4.2 Samplers and Active Texture

OpenGL actually allows several textures to be bound simultaneously. There is the idea of an "active texture," which, like everything else, is part of the state. Active texture slots are enumerated from 0 up, usually to around 8. `glBindTexture` binds to the current active texture slot. The active texture can be changed with `glActiveTexture`. This will come in handy when you need to pass multiple textures to your shader simultaneously.

Textures may only be passed to shaders in a very specific way. A 2d texture handle has type `sampler2D` (`sampler2DRect` for texture rectangle). A sampler must be a `uniform` variable, so it must be set by the application. In the shader, use the `texture*` family of functions to do texture lookups.

When setting the sampler using `glUniform`, you do not use the OpenGL texture handle received from `glGenTextures`. Instead, you use the value of the active texture to which the texture is bound. For example:

```
... // some declarations and initialization
// set active texture to slot 0
glActiveTexture( GL_TEXTURE0 );
// bind to slot 0
glBindTexture( GL_TEXTURE_RECTANGLE_ARB, tex_handle_a );
// set active texture to slot 1
glActiveTexture( GL_TEXTURE1 );
// bind to slot 1
glBindTexture( GL_TEXTURE_RECTANGLE_ARB, tex_handle_b );
```

[2] This isn't really true on newer cards; only older cards require it. However, even on some new cards, non-power-of-two-sized textures are much slower, to the point of unusability. Best to play it safe.

[3] Though most of the docs don't mention this, you can use `glTexImage2d` for texture rectangles. Just pass `GL_TEXTURE_RECTANGLE` as the first argument, instead of `GL_TEXTURE_2D`.

```
// set uniform to tex_handle_a
glUniform1iARB( texture_a_loc, 0 );
// set uniform to tex_handle_b
glUniform1iARB( texture_b_loc, 1 );
```

3.4.3 Rendering a Buffer to a Texture

In order to use the results of a previous rendering pass, you must create a texture whose data is the buffer of the first rendering pass. OpenGL provides several methods to do this of varying complexity and performance. Any should be suitable for this assignment as long as it can run in real time.[4]

The simplest method is to use `glReadPixels` and similar functions to read the buffer directly into CPU memory. Then you can load this data into a texture the same way one would create a texture from image data, using `glTexImage2d`. This is both the easiest to get working and the most well-supported by GPUs.

A more efficient and newer method involves Framebuffer Objects. A Framebuffer Object (FBO) is essentially a handle to buffer on the GPU to which rendering occurs. OpenGL provides the function `glFramebufferTexture2D` to bind a texture to an FBO, and thus the results of the render pass go directly into texture memory. This method, while certainly much faster, is somewhat more complicated and not supported on older GPUs. Note that this is an extension, and most functions will be appended with `EXT`. Consult the Lecture 7 slides, Red Book and online resources for more information.

Other methods exist, but these are the 2 recommended ones. The first is by far the simplest, while second is much faster yet still relatively simple. But you may use others if you wish.

[4] "Real time" roughly means at least 15 FPS, when compiled with optimizations. Remember that you need to build with `make MODE=release` to build with optimiztions.