15-462 Project 3: OpenGL Shader Programming

Release Date: Thursday, October 6, 2011

Project Due: Thursday, October 27, 2011, 23:59:59

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/

1 Overview

In this lab you will learn about shader programming. The lab consists of rendering a scene, similar to previous labs, using OpenGL. However, in this lab you will render two new effects that requires shader programming. Specifically, you will render outlines of the objects in the scene, which is done using a multi-pass rendering method and shaders. You will also use shaders to simulate motion blur effects whenever you move your camera.

This lab requires a firm understanding of OpenGL textures and the OpenGL Shader Language (GLSL). The OpenGL Programming Guide and GLSL quick reference chart are both very useful tools. Additionally, the OpenGL Shading Language book or "Orange Book" may also be useful (although acquiring a copy is not necessary to complete this assignment). The LightHouse 3D and TyphoonLabs (refer above) also have fairly comprehensive tutorials on shaders.

2 Description

This lab has two distint parts. The first part is about rendering outlines, similar to a cartoon drawing. You will take a standard rendering of a scene using Blinn-Phong shading and add black outlines around the geometry. Outlines are an

example of non-photo-realistic rendering (NPR). They do not occur in real life, but have uses in graphics in visualizing things. They can be used to create a cartoony, hand-drawn effect, or increase clarity in a visualization by highlighting edges of objects. However, the standard fixed-functionality pipeline cannot support such rendering, so you will have to take advantage of the programmable hardware features of the GPU.

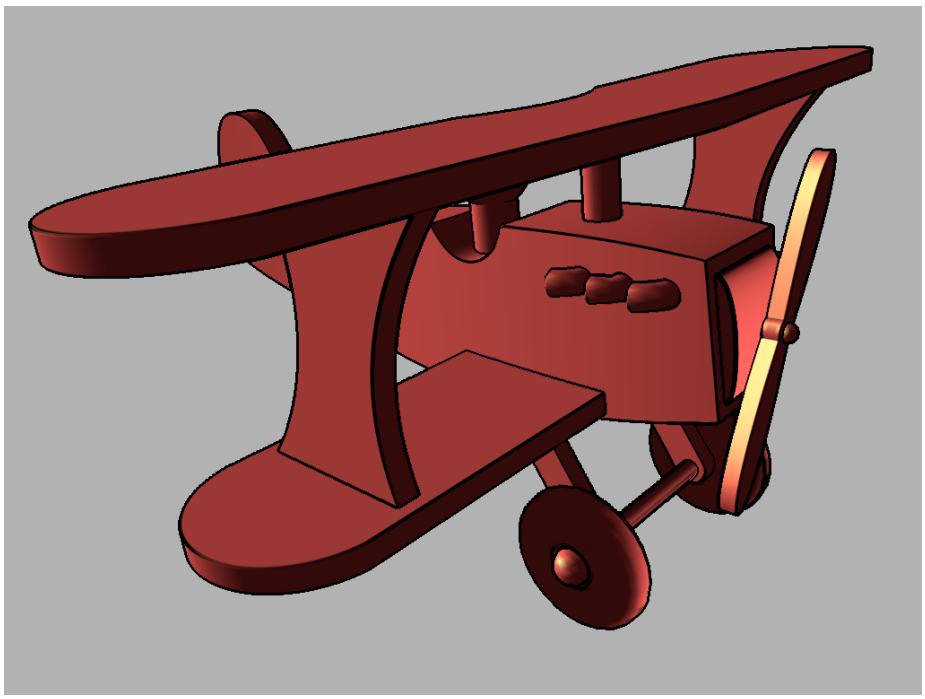

Figure 1: Example rendering with outlines. This is not a reference shot, just an example. Your renderings will differ.

The second part is about simulating motion blur effects when you move the camera. In the real world, you might see the motion blur effects when you are taking pictures of moving objects. It happens when the image being taken moves during a single shutter speed of the camera. Normally, your camera won't be moving, and only the objects are moving. However, in this lab, we will be doing the other way round, i.e. your camera's rapid movement causes motion blur while your objects stay static (since we are not doing any animation in this lab).

2.1 Global Rendering

If you remember, OpenGL's rendering is object-based, meaning the rendering of a single pixel is independent of surrounding pixels. In fact, OpenGL throws away

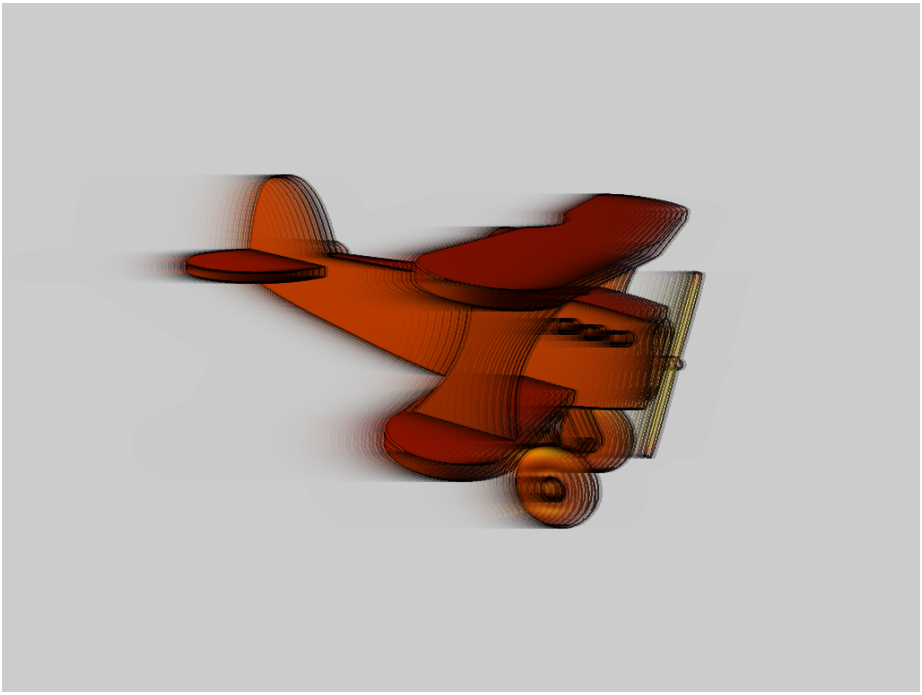

Figure 2: Example rendering with motion blur. This is not a reference shot, just an example. Your renderings will differ.

all intermediate data for each pixel immediately after rendering. Generating outlines, however, is an example of global rendering. You need information about more than one pixel to decide whether to place an outline at a given pixel.

So how can we accomplish this task with OpenGL? We will use multiple rendering passes to accomplish the goal. Only the final rendering pass will produce an output image. The earlier passes all store intermediate data needed for later passes. The output images of the earlier passes will be used in later passes as textures.

2.2 Shaders

The fixed-functionality only gives us a few pieces of output data by default, such as the depth buffer and color buffer. While we'll need both of these, we need other intermediate information. Furthermore, fixed-functionality won't let us use our intermediate information in the way we need.

So we will have to ditch fixed-functionality and program the GPU hardware with what are called *shaders*. Certain stages of the rendering pipeline aren't actually fixed and can do much more general operations. Shaders are the programs

that can be run on these stages.

There are 2 shaders you will use. The *vertex shader* runs once for each vertex, transforming the attribute data to compute, among other things, the transformed position of that vertex. The *fragment shader* runs once for each pixel of each primitive, computing the final color of the fragment. It is called a fragment rather than a pixel since a pixel can have multiple fragments, each from a different primitive, which can all impact the final pixel color. More details are in section 3 of checkpoint handout.

2.3 Generating Outlines with Shaders

2.3.1 Edge Detection and Outlines

Creating a basic outline is a fairly straightforward process, involving simple edge detection. Outlines can be computed on any array of pixels, like an image. In our case, we'll be computing outlines on a pixel buffer on the video card. But it works the same for any case.

Edge detection is an example of a convolution filter. We can think of a convolution filter as a function that operates like a sliding window over the pixels of an image. For each pixel, we compute its value as linear combination of the neighboring pixels.

In the case of edge detection, we could break this into two convolutions: one for horizontal edges and one for vertical edges. To compute the amount of edge at a given pixel, we take the difference of the neighboring pixels. For horizontal edges, this is the difference between a pixel's upper and lower neighbors. For vertical edges, we look at a pixel's left and right neighbors. The magnitude of the edge will then be the sum of the magnitudes of the horizontal and vertical edge values. For example, suppose I is our image, and (j, k) is the index of the current pixel. Equation 1 would compute the magnitude of the edge at that pixel.

$$edge_{j,k} = |I_{j+1,k} - I_{j-1,k}| + |I_{j,k+1} - I_{j,k-1}| \tag{1}$$

Note, however, that there are multiple algorithms for measuring the amount of edge, and you are not constrained to any one in particular.

To get an outline, we choose a cutoff value, and consider every pixel with an edge value beyond this cutoff to be part of the outline. We then modify those pixels in the original image to look like an outline. For example, make them all solid black.

Convolution filters are easily implemented in a fragment shader, since the shader runs once for each pixel. It can use textures to look at neighboring pixels and compute and edge value, and then modify the current pixel if necessary.

2.3.2 Creating the Outline of a Scene

The only remaining question is to decide which buffer on which to do edge detection. Doing it on the color buffer won't give us quite the results we want, since we want to outline the actual objects, not different colors.

The depth buffer makes a good candidate, since the boundaries between object/background and object/object will have a big difference in the depth, and one object will only have smooth transitions in depth and thus get no outline.

But that's not quite enough. You'd also want an outline, say, at the corner of a wall, but there is no difference in depth there. What is very different at those points are the surface normals. So we can compute the difference in angle between surface normals of each pixel to generate a second outline, and composite them together.

2.3.3 Implemented as a Shader

This leads to a multi-pass algorithm for generating outlines. You need to produce 3 different buffers: a color buffer of the actual rendering, the depth buffer, and a buffer containing the normals. This can be done in one or more rendering passes using the fixed-functionality and shaders.

Next you have to gather those buffers and store them as textures. You will be doing this in Checkpoint 1.

You then do a final rendering pass in which a shader uses all of these buffers to generate an outline and composite the final outlined image.

2.4 Motion Blur effect

Here, we will be using the naive way to generate motion blur effect, by compositing color buffers from two consecutive frames. Hence, when the camera is not moving, we will be combining two same images, and thus our model will appear static. However, when the camera is moving, at each instance of time, the current frame being rendered will be different from the previous frame rendered. So, instead of rendering the current frame, we render the combination of the two frames, and each pixel in our current frame becomes:

$$C_p(frame_i') = W * C_p(frame_i) + (1 - W) * C_p(frame_{i-1}) \tag{2}$$

where $C_p(frame_i)$ is the color of the pixel p at frame i, and W is the weight chosen to control whether the current frame or the previous frame dominates the final output. W should be a weight from 0 to 1, inclusively. Different W will result in different output, so feel free to play around with different W's and choose the one with best visual output. Any reasonable choice should be fine.

2.4.1 Implementations

In order for the techniques described above to work, while rendering $frame_i$, you will need to use the color buffer from the previous frame, $frame_{i-1}$, in your program. Thus, you should store the $frame_{i-1}$ as a texture in your program and composite both $frame_{i-1}$ and $frame_i$ You should also be careful about the first frame, since there is no previous frame. So, you might just want to use

the first frame as the final output when you first render the model, and only start using the techniques specified above after the first frame.

Note that our naive motion blur effect only takes two frames in determining the final scene rendering. This is insufficient to create a realistic motion blur effect. Thus, when rendering $frame_i$, instead of storing the original $frame_i$ (obtained from the outline shaders), you should store the new $frame_i$ resulted from the combination of $frame_i$ and $frame_{i-1}$. This will allow the "propagation" of motion from more history frames.

You are encouraged to come out with improvements to create finer motion blur effects. Good efforts might lead to extra credits.

3 Submission Process and Handin Instructions

Failure to follow submission instructions will negatively impact your grade.

1. Your handin directory may be found at `/afs/cs.cmu.edu/academic/class/15462-f11-users/`*andrewid*`/p3/`. All your files should be placed here. Please make sure you have a directory and are able to write to it well before the deadline; we are not responsible if you wait until 10 minutes before the deadline and run into trouble. Also, remember that you must run `aklog cs.cmu.edu` every time you login in order to read from/write to your submission directory.

2. You should submit all files needed to build your project, as well as any textures, models, shaders, or screenshots that you used or created. Your deliverables include:

   - `src/` folder with all `.cpp` and `.hpp` files.
   - Makefile and all `*.mk` files
   - `writeup.txt`
   - Any models/textures/shaders needed to run your code.

3. Please **do not** include:

   - The `bin/` folder or any `.o` or `.d` files.
   - Executable files
   - Any other binary or intermediate files generated in the build process.

   Run `make clean` before submitting. If you were using Visual Studio, be sure to clean the solution before submitting.

4. Do not add levels of indirection when submitting. For example, your makefile should be at `.../`*andrewid*`/p3/Makefile`, **not** `.../`*andrewid*`/p3/myproj/Makefile` or `.../`*andrewid*`/p3/p3.tar.gz`. Please use the same arrangement as the handout.

5. We will enter your handin directory, and run `make clean && make`, and it should build correctly. **The code must compile and run on the GHC 5xxx cluster machines**. Be sure to check to make sure you submit all files and that it builds correctly.

6. The submission folder will be locked at the deadline. There are separate folders for late handins, one for each day. For example, if using one late day, submit to `.../`*andrewid*`/p3-late1/`. These will be locked in turn on each subsequent late day.

4 Required Tasks

A very general overview of the implementation requirements is as follows. Refer to subsequent sections of the handout for more details.

Input: We provide you with a function to render a scene, and we provide an example shader which is not part of the scene rendering.
Output: You must use shaders to modify this rendering to have black outlines. There are no specific requirements on how exactly they look, other than they must be based on a combination of difference in depth and difference in surface normal. You are also free to choose how exactly edges are measured. You must also use shaders to simulate motion blur effects when the camera moves. Again, there are no specific requirements of how exactly it should look, other than that whenever the camera is moved, we can see some kind of motion blur effects on the objects.
Requirements:

- Your program must work on all scenes we give you, in the `scene/` folder.
- Generate an outline based on differences in fragment depth.
- Generate an outline based on differences in surface normal direction.
- Composite these outlines onto the original scene.
- Simulate motion blur effects whenever the camera is moved with keyboard or mouse.
- Submit a few screen shots of your program's renderings.
- Fill out `writeup.txt` with details on your implementation.
- Use good code style and document well. We *will* read your code.

At a minimum, you must modify `project.cpp` and `project.hpp` in the folder `glsl/`, create your shaders in the folder `shaders/`, and update `writeup.txt` to describe your techniques to do edge detection and motion blur, though you may modify or add additional source files. `writeup.txt` should contain a description of your implementation, along with any information about your submission of which the graders should be aware. Provide details on which methods and algorithms you used for the various portions of the lab. Essentially, if you think the grader needs to know about it to understand your code, you

should put it in this file. You should also note which source files you edited and any additional ones you have added.

Examples of things to put in `writeup.txt`:

- Mention parts of the requirements that you did not implement and why.
- Describe any complicated algorithms used or algorithms that are not described in the book/handout.
- Justify any major design decisions you made, such as why you chose a particular algorithm or method.
- List any extra work you did on top of basic requirements of which the grader should be aware.

There is also opportunity for up to 10% extra credit by implementing things above the minimum requirements. See section 8 for more details.

5 Starter Code

It is recommended that you begin by first reviewing the starter code as provided. Most of it is the same as the previous project. The `README` gives a breakdown of each source file. As before, you mainly need to care about `glsl/project.hpp` and `glsl/project.cpp`.

We've added a lot of new files this time. Most of these can be ignored. Almost all the new code involves the scene rendering we provide you. It will become more important in the raytracing lab, since it is the same scene format you must render with raytracing. For now, however, you don't need to bother reading/editing any of it. Everything you need is declared in `project.hpp` and the `math/` folder.

There is also an example shader enabled in the starter code by default. It is not part of the scene rendering, but merely there to provide example code for creating and using GLSL shaders.

5.1 Building and Running the Code

The code is designed to run and build on the Gates 5xxx machines and comes with a makefile. Consult the `README` for more detailed build and running instructions.

We have also provided a Visual Studio 2008 solution (works in Visual Studio 2010 Express too) , though it will take a bit of effort to get working since the programs have required command-line arguments. Note that there are differences in the compilers and graphics card across machines and thus the Windows solution might not be thoroughly tested for all possible machines. More details are in the `README`. If you use Windows, your project still **must** build and run on GHC Linux machines, so you will still have to test it on them before submitting. There are some differences in the compilers, so **code that compiles and works with Visual Studio may not compile or run correctly with**

GCC. Make sure you test it well before the deadline. Be sure not to submit Windows binaries, either.

Note that since this project takes advantage of newer GPU technologies, not all computers will be able to run GLSL shaders or some of the other OpenGL technologies you may need. Any computer with a dedicated graphics card from the last 3 years should be fine. The GHC Linux clusters and Wean Linux clusters should be fine. However, you must do all your final testing at GHC 5xxx clusters before you submit as grading will be done using the cluster machines.

Note that if you use your own laptop/computer to develop the project, subtle driver bugs and differences may cause your shader to not compile or behave differently on the school machines. Be sure to test on the school machines.

5.2 What You Need to Implement

`project.cpp` contains some empty shell functions for you to fill in. At a minimum, you should implement the initialize, destroy, and render functions. Documentation for each function is in the source file.

Feel free to modify any existing code or add new files, as long as you do not break the behavior of the program.

6 Grading: Visual Output and Code Style

Your project will be graded both on the visual output (both screenshots and running the program) and on the code itself. We will read the code.

In this assignment, part of your grade is on the quality of the visuals, in addition to correctness of the math. So make it look nice. Extra credit may be awarded for particularly good-looking projects.

Part of your grade is dependent on your code style, both how you structure your code and how readable it is. You should think carefully about how to implement the solution in a clean and complete manner. A correct, well-organized, and well-thought-out solution is better than a correct one which is not.

We will be looking for correct and clean usage of the C language, such as making sure memory is freed and many other common pitfalls. These can impact your grade. Additionally, we will comment on your C++-specific usage, though we will generally be more lenient with points. More general style and C-specific style (i.e., rules that apply in both C and C++) will, however, affect your grade.

Since we read the code, please remember that we must be able to understand what your code is doing. So you should write clearly and document well. If the grader cannot tell what you are doing, then it is difficult to provide feedback on your mistakes or assign partial credit. Good documentation is a requirement.

7 Implementation Details

7.1 Storing Information in Buffers

Most of the buffers and textures in OpenGL clamp values to be floats in the range $[0, 1]$ by default. This is true of both the depth buffer and color buffers. So you have to be a bit creative about storing data in them.

7.1.1 The Depth Buffer

The depth buffer will be filled automatically by the fixed-functionality rendering. So all you have to do is get the depth buffer into a texture and use it. The depth buffer has its own special format that differs from a typical color buffer. You will need to use the `GL_DEPTH_COMPONENT` format for the texture. For FBOs, the buffer attachment is `GL_DEPTH_ATTACHMENT`.

The depth buffer stores the z value with which the pixel was rendered. OpenGL will take care of converting z values into the depth buffer, but your shader will need to convert it back. This conversion depends on the near and far planes; the far plane maps to a depth value of 1.0, and the near plane maps to a value of 0.0. Equation 3 shows how to convert from the depth buffer value, d, to the original distance, z, where n is the near plane distance and f is the far plane distance.

$$z = \frac{nf}{f - d(f - n)} \tag{3}$$

7.1.2 The Normal Buffer

Since the fixed-functionality discards the normals, you'll have to write a shader to store the normals in the color buffer, which will then be used as a texture in the next rendering pass.

A normal is only 3 values, so we can fit it into a normal 4-byte color buffer, where each component gets one byte. For normals, each component is in the range $[-1, 1]$, so you'll have to make a simple bijection to squeeze them into $[0, 1]$ on the first rendering pass and then extract the original normal on the second.

Note that not every pixel will have a normal (e.g. areas with background color). Since we give you the rendering function, you don't get to pick the background color, but that shouldn't be a big issue, since all background pixels should map to the same normal, and background edges will be covered by the depth outlines anyway.

7.2 Suggested Sequence

We suggest you implement the assignment in the following order:

1. Continue your work from checkpoint 1.
2. Modify the shader to add outlines based on the depth buffer texture.

3. Write another shader to copy the normals into a third buffer. This may require rendering the scene using fixed-functionality twice.
4. Modify the original shader to compute outlines based on normals as well.
5. Create new shaders that take in the outlined scenes to simulate motion blur effect.

8 Extra Credit

Any improvements, optimizations, or extra features for the project above the minimum requirements can be cause for extra credit, up to 10%. Particularly impressive projects may be eligible to win a prize. Extra credit is generally awarded for impressive achievements beyond the project requirements, at the discretion of the graders.

Ideas may include but are not limited to:

- Have outlines change thickness depending on distance from the camera.
- Make a smooth transition from line to no-line instead of a sharp cutoff.
- Make the outlines smooth by implementing some kind of anti-aliasing.
- Write additional shaders to do interesting effects to the scene.
  Warning: This does not mean copying some shader you find online and fiddling with it a little. We will only accept original, interesting shaders. So no basic stuff like toon shading; it has to be pretty impressive.
- Create additional, interesting scenes (even adding new kinds of geometry).
- Add animation and create an animated scene.

9 Words of Advice

9.1 General Advice

- As always, start early. This lab takes more time than previous ones.
- Make sure you have a firm understanding of textures before starting work.
- Familiarize yourself with the rendering pipeline and where vertex and fragment shaders fit in.
- Have a look at some example shaders to see how they operate.
- Particularly since you don't have a good debugger, start with very simple shaders. Add new features iteratively, testing at each step. This will help you find shader bugs more quickly and painlessly.
- Be careful with memory allocation, as too many or too frequent heap allocations will severely degrade performance.

- Make sure you have a submission directory that you can write to as soon as possible. Notify course staff if this is not the case.
- While C has many pitfalls, C++ introduces even more wonderful ways to shoot yourself in the foot. It is generally wise to stay away from as many features as possible, and make sure you fully understand the features you do use.