COMS W4160— Computer Graphics Spring 2017

Programming Assignment 2

Out: Feb. 27

Due: 10pm, Mar. 12 (before the Spring break)

To motivate the early start and finish of the assignment, if you submit the assignment by **10pm Mar. 10th** you will receive 5% bonus points. The submission time will be determined by the timestamp of our **last** submission update on CourseWorks.

After you are familiar with OpenGL, this assignment should be straightforward. You will apply what you learned about the graphics pipeline and implement several types of shading models using the GLSL shading language. Shader programming is widely used in high-performance interactive graphics applications such as video games and VR displays. In this assignment, you need to implement algorithms for the vertex and fragment processing stages in order to achieve different shading effects.

1 Principle of Operation

As discussed in class, the *graphics pipeline* is a sequence of processing stages that efficiently transforms a set of 3D *primitives* into a shaded rendering from a particular camera viewpoint. The major stages of the pipeline include:

1. **Application** holds the scene being rendered in some appropriate data structure, and sends a series of primitives (only triangles, in our case) to the pipeline for rendering.

2. **Vertex processing** transforms the primitives into screen space, optionally doing other processing, such as lighting, along the way. The specific operations can be customized by vertex shaders.

3. **Rasterization** takes the screen-space triangles resulting from vertex processing and generates a fragment for every pixel that's covered by each triangle. Also interpolates parameter values, such as colors, normals, and texture coordinates, given by the vertex processing stage to create smoothly varying parameter values for the fragments. Depending on the design of the rasterizer, it may clip the primitives to the view volume.

4. **Fragment processing** processes the fragments to determine the final color for each, to perform z-buffering for hidden surface removal and to write the results into the *frame buffer*.

5. **Display** shows the contents of the frame buffer so the user can see them.

The detailed functionality of both vertex processing and fragment processing stages can be customized by providing the program your shader code. In this assignment, you are required to write your own shaders to satisfy the assignment requirements.

2 Getting Started

The template code for this assignment closely follows the solutions to the previous assignment, with some slight modifications to facilitate the development of multiple shaders. The compilation of the starter code is the same as in the first programming assignment. Because of the versatility and popularity of GLSL shading language, there are numerous online resources. In particular, we will use GLSL version **3.30** for testing and

grading your submissions. Please refer to the online documentation for detailed features, keywords, and reserved variables of GLSL 3.30. In addition, the online tutorial of LWJGL also gives a good example of customizing the shaders in LWJGL.

In the folder `src/resources/shaders/`, there are a two simple shaders provided as examples.

3 Programming Requirements

3.1 Guidelines

Please see the `Renderer.init()` and `Renderer.create*Shader()` methods, which will serve as a template for how to instantiate, compile, and utilize your own shaders. After you finish your implementations of shaders, you need to add a few lines of code in `Renderer.java` to create them and use them in the code.

When initializing your shader program, please see the `ShaderProgram` class (in `ShaderProgram.java`) for the methods available to you. Ensure that your shader is given a unique key and properly added to the `shaderProgramList`.

When running the template code, `Space` can be used to toggle between shaders, or directly with the following number key mapping `3`→0th shader added to list, `4`→1st shader, `5`→2nd shader.

For texture loading, please use the utilities provided in the new Texture class. We have provided easy loading of PNG files, though it is left as an exercise to make this Texture class interact nicely with your shader. **Hint:** what information needs to be passed to your shader and how can it be referenced/indexed?

In `Renderer.Render()`, you must pass the necessary updated data to your shader. A template has been provided for determining the active shader. Please see the existing examples for reference.

3.2 Required Shaders

You will implement vertex and fragment shaders to provide one *required* shading effects and a *subset* of the following kinds of shading with support for multiple light sources:

1. **Gouraud shader (required):** you are required to implement a basic Gourand shader as discussed in class. The vertex color should be computed using a Phong reflection model also as discussed in class. The color inside a triangle should be interpolated.

2. **Texture-modulated Smooth Shader (required):** Each triangle is shaded using a color value from either of the above shaders and multiplying the resulting color value by the current texture. The texture coordinate (given by `gl_TexCoord[0].st` in `GLSL`) for each vertex are used to index into the texture.

3. **Checkerboard texture (optional):** Using GLSL, you can apply a checkerboard pattern as a simple procedural texture on an object surface . You can implement it in your shader and combine it with your Gourand or Blinn-Phong shading model to show checkboard patterns (see Figure 1).

4. **Wireframe texture (optional):** You can implement a shader to highlight the mesh wireframes together with color shading. For example, it could look like a Blinn-Phong shader with the triangle edges highlighted by red color (hint: use the barycentric coordinates to determine how far a point is away from a triangle edge).

5. **Normal map (optional):** Load a image and use its 3 color channels to specify the geometry normals. In your shader, you can use the normals specified by textures to modulate shaded surface colors.

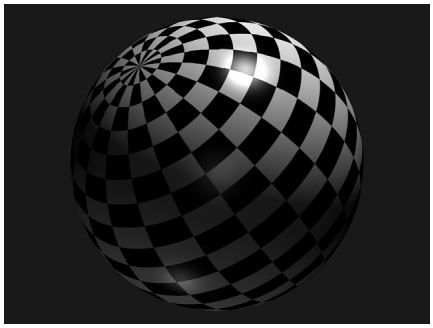

Figure 1: **A checkerboard pattern:** Combination of a checkerboard pattern and a Blinn-Phong shading model can be implemented using GLSL shaders.

6. **CEL (Toon) Shader (optional):** As I talked about in class, CEL shading is a type of non-photorealistic rendering to make 3D objects appear to be flat by using less shading color instead of a shade gradient. You can find a detailed introduction of the shading algorithm online.

You are required to implement the first two shaders (#1 and #2) above. For the remaining four optional shaders, you should implement **three** of them. Also as a hint, for #3 and #5, you will need texture coordinates of a triangle mesh. You can prepare the texture coordinates in 3D software such as Blender or for simple shapes even generate them on the fly.

3.3 Provided Shaders

We have provided two shaders

1. The **Phong** shader from last assignment
2. A **skeleton** shader with very basic behavior (adding a red color to all visible objects).

Some extraneous functionality has been added to the Phong shader example in order to give an idea of how to interact with textures. These two shaders should serve as a good starting point towards writing more complex shaders. All shaders should be comprised of a uniquely named vertex and fragment shader written in GLSL, located under `resources/shaders`. Any added textures or models should be loaded in their respective resource directories.

3.4 Bonus Requirement

You are welcome to implement other cool shading effects. Use your imagination and creativity to develop any cool effects. For example, there are details online about the Hatching and Gooch Shading effects. Make sure in the report to describe what your shader is doing. If you are inspired by some online shading effects, please add the references in your report. Please **do not** copy any code online.

3.5 Other Requirements

You are free to load any 3D models to visualize. However, in order to show the smoothly changing color effects produced by shaders, you probably need to use a smooth 3D geometry. A simple cube won't work. The focus of this assignment is on shaders. Therefore, you are allowed to reuse your code from the previous programming assignment.

4 Submission and FAQ

Submission Checklist: Submit your assignment as a zip file via courseworks. Your submission must consist of the following parts:

1. **Documented code:** Include all of your code and libraries, with usage instructions. Your code should be reasonably documented and be readable/understandable by the TAs. If the TAs are not able to run and use your program, they will be unable to grade it. Try to avoid using obscure packages or OS-dependent libraries. To ensure the TAs can grade your assignment, it is highly suggested to include details of compiling and running your code.

2. **Brief report:** Include a description of what youve attempted, special features youve implemented, and any instructions on how to run/use your program. In compliance with Columbia's Code of Academic Integrity, please include references to any external sources or discussions that were used to achieve your results. *I highly suggest you put a few pictures in your report to illustrate your shading effects.*

3. **Video highlight!:** Include a video that highlights what youve achieved. The video footage should be in a resolution of 960×540, and be no longer than 10 seconds. We will concatenate some of the class videos together to highlight some of your work.

4. **Additional results (optional):** Please include additional information, pictures, videos, that you believe will help the TAs understand what you have achieved.

Evaluation: Your work will be evaluated on how well you demonstrate proficiency with the requested shader code, and the quality of the submitted code and documentation, but also largely on how interesting and/or creative your overall effects is.

Please don't copy the shader code from somewhere else. We are going to run automatic tools to detect plagiarism when we grade. If there is a significant similarity, you will be required to explain it.