Assignment 3: Geometric Modeling

CS 148 Autumn 2019

Due Date: Monday, 14 *October 2019 by 7pm*

Follow the instructions carefully. If you encounter any problems in the setup, please do not hesitate to reach out to CAs on Piazza or come to office hours. Start early!

1 Update Your Code

Go to *main.cpp* and change the line that says:

```
#define APPLICATION Assignment2
```

to

```
#define APPLICATION Assignment3
```

You will be making changes primarily within the "Assignment3.cpp" file.

2 Introduction

In this assignment, you will become familiar with the process of taking a piece of geometry, whether it be one you found or one you created, and placing it into your scene. This is the first step towards creating your final scanline image. We will be using the Open Asset Import Library to load meshes, so make sure the meshes you find and create are supported. In particular, it is highly recommended that you use the OBJ file format. It is also highly recommended that you become familiar with the OBJ file format. Information is freely available online and this website is particularly useful. If you have not yet gotten the assignment framework to compile, please contact a CA ASAP. **Note that setting up assimp was part of Assignment 1; so it should already be compiled and linked for you!**

2.1 Finding Meshes Online (For Free!)

- TurboSquid.
- PolyCount. Note that it may be useful to Google "Polycount free models".
- TF3DM.

You may find it useful to look for people who have used "Zbrush" to create high-poly models and then trying to find a way to get an OBJ from that.

2.2 Creating your Own Meshes (For Free!)

- Autodesk Maya
- Autodesk 3ds Max

- Blender

Additionally, you can generate your models procedurally in code. We recommend you spend time becoming familiar with your 3D modeling package of choice.

3 Requirements

In this assignment you are required to setup part of your scene for the final project image. **You are required to have, at the very minimum, two models in your scene**. **One of them should be found online** (for example, on the websites that we have provided) **and the other should be created by yourself** either using software or with code. Do not play with the lights or the shader for this assignment. Note that you may have to tweak the camera parameters to make your models visible.

3.1 Background

For this assignment, you will be using the provided framework's abstractions over OpenGL and shaders instead of writing everything yourself as in Assignment 2. If you want to know what a particular class or function does, we have provided documentation that you can find by opening "doxygen/html/index.html" in your favorite web browser. If you want to learn the OpenGL boilerplate that gets abstracted away, feel free to read the source code and ask questions on Piazza! Inside the Assignment3 class, there are two functions that are used to load the scene: `SetupExample1` and `SetupExample2`. These functions do three things:

- Load and setup shader(s) with the right material parameters.
- Load a mesh and add the mesh to the scene.
- Load light(s) and add the light(s) to the scene.

The shader and light setups can be used as-is for the assignment this week—we will examine them further next week. `SetupExample1` uses the function `PrimitiveCreator::CreateIcoSphere` to demonstrate how to create a shape programmatically. `SetupExample2` uses the function `MeshLoader::LoadMesh` to demonstrate how to load a mesh using the assimp library. By default, `SetupExample1` is called to create the scene that is initially displayed. The `HandleInput` function will call `SetupExample2` if the "2" key is pressed. You can look at the implementation of the RenderingObject class in "common/Rendering/RenderingObject.cpp" to connect your knowledge of vertex buffers in OpenGL from Assignment 2 with the framework. Another important function to look at will be the *SetupCamera* function which creates a camera by giving it the field of view (the default is set very high) as well as the aspect ratio. Functions for setting the near and far clipping planes can be found in the PerspectiveCamera class ("common/Scene/Camera/PerspectiveCamera.h"). Many people show up with "invisible" models because their cameras were placed improperly for the meshes they have loaded. Generally, this means that the camera is inside the model or the model is not within the view frustum.

3.2 Model Location

Models are stored in the "assets" folder. When the program is configured and compiled, a C++ preprocessor definition is added to tell the program where to look for this folder. When you load a model, make sure the path is specified relative to the "assets" folder. That is, if you find an ".obj" file you want to use, e.g. "myFavoriteMesh.obj" and store it as "assets/aFolder/myFavoriteMesh.obj," the filepath you would use to find that OBJ file in the assignment framework is "aFolder/myFavoriteMesh.obj."

4 Cool Things to Do

Here are some suggestions to go beyond the bare minimum necessary to complete this assignment. Only attempt to do these once you have met the requirements for the assignment!

4.1 Subdivision

Take the loop subdivision concept from class and implement it. You may find it helpful to look at the "PrimitiveCreator::CreateIcoSphere" function in "common/Utility/Mesh/Simple/PrimitiveCreator.cpp" which will iteratively subdivide an icosahedron. Note that it may be useful to introduce the concept of "Triangles" into the code to accomplish this.

4.2 3D Scanning

A recent trend is the usage of RGB-D cameras to scan an object in real life to create a 3D model. If you have access to one, you can use the Structure Sensor. Alternatively, free structure from motion software such as VisualSFM can be used to reconstruct geometry from multiple images. You may find it useful to process your mesh using MeshLab to remove any free-floating parts of the mesh, to close the mesh, or to otherwise refine the scanned model. Your scanning application will probably give you vertex colors or a texture for the mesh. You will want to store all this information for use in a later assignment should you choose to use your mesh for the final scanline image.

5 Behind the Scenes (Recommended Reading)

This section is for you to better understand what is being abstracted away from you by the assignment framework. You will find this helpful in trying to debug and understand what is happening.

5.1 Mesh Loading

To load a mesh, you will call "MeshLoader::LoadMesh" with a shader object and the path to the mesh to load. As noted earlier, this path must be relative to the "assets" folder. For now, you can just use the a default shader and you can load it in using the following code (this is the exact same code already in the "Assignment3::SetupExample1").

```
#ifndef DISABLE_OPENGL_SUBROUTINES
    std::unordered_map<GLenum, std::string> shaderSpec = {
        { GL_VERTEX_SHADER, "brdf/blinnphong/frag/blinnphong.vert" },
        { GL_FRAGMENT_SHADER, "brdf/blinnphong/frag/blinnphong.frag"}
    };
#else
    std::unordered_map<GLenum, std::string> shaderSpec = {
        { GL_VERTEX_SHADER, "brdf/blinnphong/frag/noSubroutine/blinnphong.vert" },
        { GL_FRAGMENT_SHADER, "brdf/blinnphong/frag/noSubroutine/blinnphong.frag"}
    };
#endif
    std::shared_ptr<BlinnPhongShader> shader =
        std::make_shared<BlinnPhongShader>(shaderSpec, GL_FRAGMENT_SHADER);
    shader->SetDiffuse(glm::vec4(0.8f, 0.8f, 0.8f, 1.f));
    shader->SetAmbient(glm::vec4(0.5f));
```

We will dig more into shaders and their relation to shading and lighting next week. If you recall from Assignment 2, when you want to draw a mesh, you have to create a VAO and create VBOs that store vertex properties to send to the GPU. All this happens inside "RenderingObject::InitializeOpenGL" and the functions it calls. The only difference is that the "RenderingObject" class will also handle various other vertex properties such as vertex normals, UV coordinates, per-vertex colors, etc. One difference though is that unlike in Assignment 2 where we let OpenGL assume that each triplet of vertices is a triangle, the rendering framework instead has the option to use the "glDrawElements" call which also passes in an "element buffer object" (the "vertexIndexBuffer") which tells OpenGL which vertices to use together to create a single face (triangle).

5.2 Transforms

As you learned in class, a vertex gets transformed using the "model" matrix M, "view" matrix V, and "projection" matrix P to transform the vertex from "object" space to normalized device coordinates. The "model" matrix transforms a point from object space to world space. The "view" matrix transforms a point from world space to camera space. The "projection" matrix transforms a point from camera space to NDC. This transformation is handled for you in the provided vertex shader in "blinnphong.vert". These two lines of code

```
vec4 vertexWorldPosition = modelMatrix * vertexPosition;
gl_Position = projectionMatrix * viewMatrix * vertexWorldPosition;
```

take in a 3D point in homogeneous coordinates x and the NDC p such that $p = PVMx$. Note that the vertex shader takes in these matrices as uniform variables

```
uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;
```

which means that the assignment framework C++ code handles passing in these values using some "glUniform" function call.

An object's transformation matrix (object space to world space transformation matrix) is stored in the "SceneObject" class ("common/Scene/SceneObject.h"). We store the translation, rotation, and scale of the transformation matrix separately in three variables.

```
glm::vec4 position;
glm::quat rotation;
glm::vec3 scale;
```

Whenever, one of these values change, the transformation matrix is re-computed using "SceneObject::UpdateTransformationMatr
You can modify these variables using the public API of the "SceneObject" class. Refer to the comments in "SceneObject.h" (or the corresponding doxygen document) for the following functions for information on what they do:

- Translate
- Rotate
- MultScale
- AddScale
- SetPosition

The transformation matrix computed using "SceneObject::UpdateTransformationMatrix()" is used as the "modelMatrix" for all the objects that you load.

Note that the "viewMatrix" needs to instead transform a point from world space to camera space. As a result, the "PerspectiveCamera::UpdateTransformationMatrix" is overridden to invert the transformation matrix which will thus compute a transformation matrix to transform a point from world space to object space (which in the case of a camera, is "camera space"). Note that the "PerspectiveCamera" class inherits from the "Camera" class which inherits from the "SceneObject" class so anyh of the functions you use to change the transformation matrix of a model you load can also be used to transform the camera. Finally, the "projectionMatrix" is computed using "PerspectiveCamera::GetProjectionMatrix." This function calls "glm::perspective" which computes a perspective projection similar to the one shown in class.

5.3 Shaders

Finally, note that all these transformation matrices need to be passed from the CPU to the GPU. This all gets taken care of in "SceneObject::PrepareShaderForRendering."

```
shader->SetShaderUniform(MODEL_MATRIX_LOCATION, GetTransformationMatrix());
shader->SetShaderUniform(VIEW_MATRIX_LOCATION, currentCamera->GetTransformationMatrix());
shader->SetShaderUniform(PROJECTION_MATRIX_LOCATION, currentCamera->GetProjectionMatrix());
```

This assumes that if the shader wants to accept the model, view, and/or projection matrix, it has a uniform of type `mat4` with the names specified in `MODEL_MATRIX_LOCATION`, `VIEW_MATRIX_LOCATION`, and `PROJECTION_MATRIX_LOCATION`. The "ShaderProgram::SetShaderUniform" abstracts away the "glGetUniformLocation" and "glUniform" call you had to do in Assignment 2.

6 Grading

This assignment will be graded on the following requirements

- There is at least one model that you found online in your scene. The "outlander" model provided with the framework will not count towards this requirement.
- There is at least one model that you created/generated in code in your scene. The sphere or plane created by the provided assignment framework will not count towards this requirement.

according to the following rubric.

- 4 – One qualifying hand-made/generated model and one qualifying downloaded model are found in the scene.
- 2 – One qualifying model is found in the scene.
- 0 – Zero qualifying models are found in the scene.

You will not get points deducted or added if you change the shader or lighting conditions of the scene.

6.1 Qualifying Models

Any model you find online or create/generate yourself must be more complex than a simple geometric primitive (e.g. sphere, cube, plane, conic, etc.) or a trivial combination of multiple geometric primitives (e.g. two spheres stacked on top of each other). You will not receive credit for any piece of geometry that can be made without non-trivial modifications in any 3D modeling package. Any non-rigid transformation (i.e. any deformation that is not just a scale, translation, or rotation of the mesh) will be considered a non-trivial modification. If these requirements are at all unclear, ask a CA for guidance and clarification. We are, however, not looking for masterpieces with any mesh that you find/create/generate.

7 Looking Ahead

Your final ray-traced image (the final project) will, at the least, require one model you made/generated as well as one model you found online. Generally, students have no problems finding free models online. For the model you create/generate yourself: although we require it to be in the scene, it does not have to be the focus of your image so do not worry too much if you do not like how your model looks.