Notes:

- Reading material: Chapters 14, 15
- Consult the GLM and OpenGL/GLSL resources and tutorials available on Moodle as needed.
- You are strongly encouraged to work in small groups of two.
- Exam is planned for Nov 9 at 2:00pm

Objectives

In this assignment, we will upgrade our infrastructure to allow for multiple types of materials with different GLSL shaders in a single scene. One particularly interesting material that you will help implement is a bump mapping material.

You are provided with an asst5 starter folder for this assignment. There are a few support files and two new shaders `normal.{vert,frag}`. To start, copy the files from your asst4 folder to your asst5 folder (solution files from asst4 are provided on Moodle and you may use them, or you may use your own solution if you are sure of it). Rename `asst4.cpp` to `asst5.cpp`. Don't add the file `asst5-snippets.cpp` to your visual studio or xcode projects.

Task 1: Material Infrastructure

So far we have been using only one GLSL shader per frame during our rendering, as controlled by the global variable `g_activeShader`, an index into the global array of ShaderState's `g_shaderStates[]`. A more complex graphics program, such as a modern game, will use quite a few shaders to model different surface properties. In this assignment, we will incorporate a material infrastructure that allows you to use multiple shaders in a single frame. The following sections explain the new classes that will be included in your program. Note that you should not start altering your code until the last sub section; *please read all the text before that section first.*

Uniforms

Conceptually, each shader takes in a bunch of uniform variables. Some of these uniform variables, such as the projection matrix, hold the same value for all shaders, and are set once at the beginning of the frame by you. Originally, you set it directly to the currently active GLSL shader. But if we start using a different GLSL shader in the same frame, we will need to set the values of these uniforms again, because different GLSL shaders do not know about the values of each other's uniform variables. This suggests we need some kind of data structure to hold the values of these uniform variables. Then when a new shader is selected, we populate its uniform variables with values from this repository.

In the supplied code, the `Uniforms` class defined in `uniforms.h` serves this purpose. It is essentially a dictionary mapping string name to uniform values. To use it , you would write

```
// Suppose uniforms is of type Uniforms, and m is of type glm::mat4
uniforms.put("uProjection", m);

// Suppose light is of type glm::vec3
uniforms.put("uLight", light);

// Set uColor variable to red
uniforms.put("uColor", glm::vec3(1, 0, 0));

// You can even chain the put, since put returns the object itself
```

```
uniforms.put("a", 1).put("b", 10).put("c", glm::vec2(1, 2));
```

As you can see the first argument of `Uniforms::put` is the name of uniform variable as you have declared in the shader code. The second argument is the actual value. Right now `put` is overloaded to take the following types as the second parameter:

- a single `float`, `int`, or `glm::mat4`

- a `glm::vec<n, T>` with `n` being 1, 2, 3, or 4, and `T` being `float`, `int` or `double`, i.e., `glm::vec*`, `glm::ivec*`, and `glm::dvec*` of various sizes.

- A `shared_ptr<Texture>` type

The last bullet needs further explanation: If you recall from assignment 2, shaders can access textures by declaring a `sampler2D` typed uniform variables, e.g.,

```
uniform sampler2D uTexUnit0;
```

It can then use the GLSL function `texture()` to look up values in the texture. However, the value that you supply to this uniform variable is an integer telling it which texture unit to use, as opposed to the actual texture handle. So before calling the shader, you need to manually bind the texture to the correct texture unit first by calling the appropriate sequence of `glActiveTexture` and `glBindTexture`.

In the provided material framework, the binding of texture to texture units is automatically done behind the scene. So to provide the value of a `sampler2D` variable in a `Uniforms` object, you simply put in a shared pointer to the actual texture, encapsulated by the `Texture` class (more on this later).

Now to recap, in the original codes, you set any uniforms directly by pulling the handle from `ShaderState`, and then calling one of `glUniform*` variants. In the new code, you will set the value to a `Uniforms` class, which will hold on to the value until it is needed by a shader.

RenderStates

Recall that when you draw the arcball, you enclose it with a pair of `glPolygonMode` calls to make it drawn in wireframe. OpenGL has a "state-machine" model; after you have change some states, all subsequent drawing commands use the new states. Hence after you have instructed OpenGL to draw things in wireframe, and drawn the arcball, you need to turn off the wireframe with another `glPolygonMode` call. This will quickly become hard to manage since different material properties might need different OpenGL state flags to be set, and your code will quickly become peppered with these state changing calls. (For example, in the next assignment, you will need to set states that controls how transparent objects are drawn. One of the state you will need to change for that assignment, is to enable *framebuffer blending*, by calling `glEnable(GL_BLEND)` before drawing the transparent objects. Afterward, you will need to call `glDisable(GL_BLEND)` to disable blending for non-transparent objects.)

To make your job easier, we have introduced a new class called `RenderStates`. It stores a subset of OpenGL states (that set will become larger and large as more states become relevant for our assignment). For example, it contains a `polygonMode` member function which takes the same arguments as `glPolygonMode`. The difference is that, when you set a state using `RenderStates`'s member function, that state does not immediately take effect in OpenGL, but is stored within the object. Later, when you want a `RenderStates` internal states to take effect in OpenGL, you will call its `apply()` member function. Even when you do not set a state changing member function such as `polygonMode`, the `RenderStates` has a default value for that particular states. Thus, each `RenderStates` stores the values of the same set of `OpenGL`states. After a `RenderStates` has been `apply`'ed, the current OpenGL states will agree with the `RenderStates`'s set of states.

So why would this be useful? Consider the following code snippets

```
// All three have a default polygonMode set to GL_FILL, and blending disabled
RenderStates r1, r2, r3;

// set r2 to be used for wireframe rendering
r2.polygonMode(GL_FRONT_AND_BACK, GL_LINE);

// set r3 to be used for transparent objects
```

```
r3.enable(GL_BLEND);

// At this point, actual OpenGL states have not been changed yet.
// Now we can switch between the three sets of render states easily

r2.apply(); // after this, GL states correspond to that of r2

// draw stuff in wireframe, and translucent;
r1.apply(); // after this, GL states correspond to that of r1

// draw stuff not in wireframe, and non translucent
r3.apply(); // after this, GL states correspond to that of r3

// draw stuff not in wireframe and translucent.
r3.apply(); // after this, GL states correspond to that of r3

// draw stuff not in wireframe and translucent.
```

As you can see, there is no saving and restoring of OpenGL states. All of that are handled by the `RenderStates` internal implementation.

Geometry and Texture

You have already encountered the `Geometry` and `GlTexture` classes before. `Geometry` encapsulates the OpenGL VBO and IBOs handles, whereas `GlTexture` encapsulates the OpenGL texture handles. So far we have had one type of `Geometry`: static geometry with a single vertex layout as defined by `VertexPN`. Similarly we have had a single type of Texture: 2D image maps. A more complex graphics program will often feature more different types of geometries and textures. For example: some geometry might need to be generated dynamically and rendered on the fly; cube texture maps (which are six 2D image textures that are mapped on an imaginary cube) might be needed to model a reflection effect; and so on.

Given that we are introducing a new Material system, and both geometry and texture need to interact with it, it is now a good time to refine our `Geometry` and `Texture` definitions a bit so that they play well with the new Material system, and can be extended to support different kind of geometries and textures in the future. The new `Geometry` and `Texture` classes are both abstract classes with certain functions that once implemented, allow custom Geometry and Texture type to be used by the Material system. You should look at the detailed comments accompanying their definition in `geometry.h` and `texture.h` for details.

To demonstrate custom types of geometry and textures, we supply one concrete implementation of Texture and a bunch of implementations of `Geometry`, in `texture.h` and `geometry.h`.

- `ImageTexture`: A `Texture` that loads a PPM file with three channels, and optionally stores the content in SRGB color space.

- `SimpleIndexedGeometry{PN|PNX|PNTBX}`: `Geometry` implementations that encapsulate an index buffer and a vertex buffer consisting of vertices that provide different vertex attributes

  - PN: Position and Normal
  - PNX: Position, Normal, and teXture coordinates
  - PNTBX: Position, Normal, Tangent, Binormal, and teXture coordinates

- `SimpleGeometry{PN|PNX|PNTBX}`: `Geometry` implementations that only encapsulate a vertex buffer with vertices of different format.

The different types of geometry classes above all use a more powerful and flexible geometry implementation called `BufferObjectGeometry` under the hood. `BufferObjectGeometry` defines a geometry object using one or more vertex streams of arbitrary format, zero or one index stream, and can draw its data as different primitives (e.g., triangles, lines, points, etc.)

`Material`: **Pulling all together**

Given the above, the actual `Material` class is a light-weight container of the following:

- It keeps a shared pointer to the actual GLSL shader program used.

- It keeps a `Uniforms` field, accessible through its `getUniforms()` member function.

- It keeps a `RenderStates` field, accessible through its `getRenderStates()` member function.

The default copy semantics works fine for `Material`, so one can make a copy a `Material` by writing codes like `Material material2 = material1`. This performs a deep-copy, so if you then change the uniforms stored in `material2.getUniforms()`, it won't affect the uniforms stored in `material1`.

The `Material` has a `draw(geometry, extraUniforms)` member function that takes in a geometry and some extra uniforms. This function will first set the GLSL shader, and set all the `RenderStates` it contains. Next it binds all uniforms used by the shader, first by looking at its own `Uniforms`, and then looking at the passed in `extraUniforms`. Textures are properly bound and set at this stage as well. Then it makes sure that the supplied `geometry` can provided the vertex attributes used by the shader, and enables the corresponding vertex attribute arrays. It then call `geometry`'s `draw` function to draw the geometry. Finally it disables the previously enabled vertex attribute arrays.

As an example, suppose you want to use the new Material system to draw the arcball as a wireframe sphere with solid color, you would do the following

```
// Assume uniforms is of type Uniforms and already contains the projection matrix
// Assume g_arcballMat corresponds to a Material with the right renderstates and uniforms set

// Calculate the arcball's MVM

// Record it in uniforms
sendModelViewNormalMatrix(uniforms, MVM, normalMatrix(MVM));

// Draw with the material, assume g_sphere points to a Geometry for a sphere
g_arcballMat->draw(*g_sphere, uniforms);
```

The constructor of `Material` takes in two arguments, the file name of the vertex shader, and the file name of the fragment shader.

Code Migration

Finally, follow the instructions in `asst5-snippets.cpp` to change your `asst4.cpp` (or the solution code provided) to use the new Material system. As you run the resulting program, you will notice that

- the cubes are drawn with diffuse shading

- the arcball is drawn in wireframe

- the ground is drawn with a texture.

Task 2: Smiley Cubes

Our cubes have so far been shaded using a diffuse fragment shader that simulates how light bounces off a rough material. Your task here is to change the appearance to be a combination of this diffuse color and a color looked up from a texture. You can use the `smiley.ppm` file from asst 2 as the texture for all six faces of the cubes.

The textures coordinates (x, y) for all vertices are built into our objects. You need to:

- modify the vertex shader `basic.vert` so it takes in a `vec2 aTexCoord` attribute and outputs a `vec2 vTexCoord` that can then be accessed in the fragment shader.

- add a `sampler2D` uniform variable `uTexColor` to the fragment shader so you can use `texture(uTexColor, vTexCoord)` to look up the needed color from the texture. The color used

in the lighting computation is then a combination the uniform color passed in (red or blue) and the color looked up from the texture.

- when the cube materials are initialized, make sure you create an `ImageTexture` and add it the set of uniform variables of those materials so it gets properly bound to the texture units.

Task 3: Bump Mapping

To make your pictures look even snazzier, you will help finishing a bump mapping implementation.

You are given a texture `FieldstoneNormal.ppm`, but the 3 values making up its pixel will not be interpreted as RGB values, but rather as the 3 coordinates of a normal. If you refer to initMaterials() that you inserted in your program, you will see that this texture is bound to the `uTexNormal` uniform variable of the `g_floorMat` material, which uses the shaders `normal.{vert|frag}`. You need to change the fragment shader `normal.frag` to read the appropriate pixel from the texture, transform it to eye space, and use it for shading calculation. This will give your geometry a high resolution look.

Texture coordinates: each vertex will need (x, y) texture coordinates. We will build these in our ground, cube, and sphere geometry objects. These attributes are passed into the vertex shaders for the draw call. You can access them as `vTexCoord` in the fragment shader.

Data range: The texture stores its pixel data as triplets (rgb) of real numbers between 0 and 1, while normal coordinates are triplets (xyz) of numbers in the range -1 to 1. Thus you need to apply a scale and then shift to the data before using it. The scale and shift should be chosen so that $0 \mapsto -1$ and $1 \mapsto 1$.

More matrix stuff: We want to store our normal map data in such a way that we can use one data patch and tile it around a curvy surface (like a sphere or a mesh). As such, we don't want to represent the normal data in the texture as coordinates with respect to the world or object frame.

For sake of notation, let $\vec{\mathbf{b}}^t = \vec{\mathbf{e}}^t M$ be the frame associated an object that is about to be drawn (the ground, in our case), and M the associated model view matrix.

We will let each vertex of this object have its own "tangent frame" represented as $\vec{\mathbf{t}}^t = \vec{\mathbf{b}}^t T$. The data for T will be passed as three `vec3` vertex attribute variables: `aTangent`, `aBinormal`, and `aNormal`. These will represent the three columns making up the upper left 3 by 3 submatrix of T. Since we will be dealing with the coordinates of vectors and not points, we will not need any translational data in T. Our convention is that the data in our bump mapping texture (after scaling and shifting to the correct data range), $\mathbf{n} = [n_r, n_g, n_b, 0]^t$, expresses the normal wrt the $\vec{\mathbf{t}}^t$ frame. Thus the object frame coordinates of the normal's coordinate vector are $T\mathbf{n}$ and its eye coordinates are $M^{-t}T\mathbf{n}$.

Since we will not get hold of the normal data until we get to the fragment shader, we will do the following: at the vertex shader we pass $M^{-t}T$ as a varying variable called `vNTMat` (for normal matrix times tangent frame matrix) to the fragment shader. Then at the fragment shader, you will multiply $\mathbf{n}$, the normal data fetched, by this matrix.

In the fragment shader, if we have the eye coordinates of a vector $\mathbf{v} = [x_e, y_e, z_e, 0]^t$ that we want to dot with a normal, we simply compute `dot(normalize(vNTMat * n), normalize(v))`.

Submission

Submit your solution on Moodle. Make sure you add a short README.txt that includes a description of what you did and any instructions we may need to run your code.