Phase 3 Help Document

In this document, we will break down the requirements of phase 3 to make it more clear.

Part 1: Texture2D and Sampler Classes

The Texture2D class should hold an OpenGL texture object. The Sampler class should hold an OpenGL sampler object. Both of these classes are responsible for creating their own OpenGL objects and deleting them automatically when the class instance is deconstructed. You will find the ShaderProgram and the Mesh classes as good points of reference. Also, these classes should comply with the RAII pattern so don't forget to handle the copy constructor and assignment operator; the simplest solution is to delete them. You will also need to adapt the texture loading functions to support your new texture class.

Part 2: RenderState Class (or Struct).

Starting from this phase, we will no longer assume that all the objects will set the OpenGL pipeline in the same manner. In other words, some objects may be transparent, some may be opaque, some objects will only be viewed from the outside (backface culling), some will be viewed from the inside and the outside (no culling), etc. So, we need to create a class or a struct that defines what OpenGL pipeline settings to use when rendering an object. These settings should include:

- **Culling**: Whether culling is on or off, which face to cull and which winding direction is to be considered the front face.
- **Blending**: Whether blending is on or off, which blending equation to use and the source and destination blending factors. Also you can add the constant blend color to the render state.
- **Depth Testing**: Whether depth testing is enabled or not, and which testing function to use.

There are some other optional settings that you can also add if you want such as stencil testing, color/depth/stencil masks and alpha to coverage.

Part 3: Material Class.

So in the last phase, we had to find a way such that the Mesh Renderer can know which shader to use. If we had any details specific to the rendering of that specific object (such as what color to send to the tint uniform variable used in the fragment shader), we had to store these details in the Mesh Renderer. However, it is common that many mesh renderers would share these data and it would be inefficient to store them in every instance. Therefore, we will create a class called Material that will store a pointer to the shader program, the values of the uniforms (other than the transformation matrix) and the render state. While storing the pointer to the shader program and the render state is simple, storing the uniform values can be tricky so we will discuss a few approaches.

The first and simplest approach is to collect all the uniforms that you plan to use and keep a variable for each of them. So if some shaders need a tint, keep a color (vec4) variable for the tint in the material. If some shaders need a texture, keep a pointer to a texture2D and sampler for it in the material and so on. The problem with this approach is that it is inflexible and you will either have to limit yourself to use only the variables you defined or you would need to add more variables to accommodate for more shaders.

The other approach is to store a dictionary (or list) of material properties. Each property will be sent to the uniform that carries the same name. However, each uniform variable can have a different type so the material property should be able to hold different types. To do this, you can either use “std::any” which is supported in C++17, or you can use polymorphism where you create an abstract base material property class and derive from it a set of material properties for each uniform type (e.g. class FloatMaterialProperty : public BaseMaterialProperty { float value; };). It will be tiresome to create a class for each property type so you can rely on template classes instead (e.g. template<typename T> class MaterialProperty : public BaseMaterialProperty { T value; };). Now you can create a dictionary of properties where the key is the uniform name and the key is either “std::any” or a pointer to a material property.

Note that for texture properties (those who map to a sampler in the shader) should store 2 pointers: a pointer to a texture 2D and a pointer to a sampler. If you use the first approach, such create 2 variables (one for the texture and the other for the sampler). If you use the second approach, you can either store an “std::pair<Texture2D*, Sampler*>” or create 2 pointers in the derived “TextureMaterialProperty” class. If you plan to use templates, you can use template specialization. Also remember that if you have more than one texture in your shader, you have to make sure that each texture is bound to a different texture unit.

Part 4: Light Component

Now we need to add light into the scene so we create a light component. This component should hold all the data related to a light including its type but excluding the position and direction which will be retrieved from the transform component.

Part 5: Support Textures, Materials, Render States and Lights in The Render System.

First, you need to make the necessary modifications to the render system such that it would work for textures and materials.
Second, we need to properly support transparency. This means that we have to sort the objects before drawing them. To know whether an object is transparent or not, check their blending state from their render state. Transparent objects must be rendered after all opaque objects and they must be sorted such that they are rendered in order from the farthest to the nearest object to the camera.
To support lights, the render system has to collect the lights from the world then send their data to the shaders.

Part 6: Texture and Light support in the Shaders.

We need to modify the shaders such that they draw the texture correctly (based on the texture coordinates stored in the Mesh) on the object. The same goes for lights. Look at the examples for a reference to how to add support for textures and lights. The last example (Example 23) will be the most useful example for this part.

Render System after phase 2 and phase 3 modifications

Before modification:

```
Some code to get the rendering camera entity.
Loop on all entities:
    ● Draw the entity using its attached mesh renderer component.
```

After modification:

```
Some code to get the rendering camera entity.
Collect all the lights
Let M be an empty container containing mesh renderers and their distance from the camera.
Loop on all entities:
    ● Calculate the distance from camera transform to the selected entity transform.
    ● Add it to M
Sort M such that:
    ● Opaque objects are before transparent objects
    ● Transparent objects are ordered from far to near
Loop on all M:
    ● Setup the Material:
        ○ Use the shader program
        ○ Send transform and camera variables to shader uniforms
        ○ Send material variables to shader uniforms
        ○ Send lights to shader uniforms
        ○ Use the render state to set openGL state
    ● Draw the entity using its attached mesh renderer component
```

Part 7: Create a scene with textures and lights.

Modify the scene from phase 2 to include lights and textures.