**Objective**

Today’s homework assignment will require you to load in, and render multiple times, a model, as well as light the scene with multiple lights. There are multiple ways to do multiple lights in a scene, but the way we’ll be doing it today is with multiple passes.

**Homework Requirements**

Features you’re expected to implement:

* Chair model is properly loaded
* Ground is properly textured and lit
* Chairs are properly textured and lit
* Point light is working properly
* Spotlight is working properly

**Homework Instructions**

**Model loading**

* The model loading will be the most complex part of this lab. You might want to try getting the lighting working properly with just the ground quad first, then once the lighting and ground are correct, moving onto the model loading.
* The model you’ll be trying to load is a chair that is in the OBJ format. An explanation of the file’s structure can be found here: <http://www.fileformat.info/format/wavefrontobj/egff.htm>
* Similar to the Exporter lab for 3DCC, the OBJ will not be optimized. So after we’ve read in the data we’ll need to optimize it by looping through all of the triangles read in, and creating a list of unique verts and a list of indices into that unique vert list, the basic outline for those who need a refresh is as follows:
  + For each of our faces
    - For each vert of that face (protip: triangles have 3 verts)
      * Loop through all our unique verts
        + If the current vert matches a vert in our unique vert list

Set the vert’s index to the index of the unique vert it matches

* + - Otherwise, if it matches NONE of the unique verts
      * Push the vert into the unique vert list
      * Then set the index in our index list to the index of the unique vert we pushed in.
* By the end of this we should have an index list of about 990 and a unique vert list of something in the mid 400 range. Looking at the old 3DCC slides or documents for the maya exporter lab might be helpful if you’re lost.
* **NOTE:** The UV coordinates of an OBJ are done differently than DirectX is expecting them. You'll need to **inverse** the Y value for your textures to show up properly!

**Ground Quad**

* You’ve made quads before, but this one will not only need texture coordinates, but normals as well.

**Directional Light**

* There will be a single BLUE directional light applied to objects in your scene with a direction of {-1, -0.5, 0}.
* A directional light is the easiest of all the lighting algorithms (not including ambient lighting), and the slides from today’s lecture should cover all you need to know about this and other lighting calculations.

**Point Light**

* You’ll be making a single GREEN point light to apply to your scene. The point light will have a position of {3, 2, 3} and a radius of 5.
* Remember, because our light has a radius, we’ll need to calculate attenuation.

**Spot Light**

* The last light in your scene will be a RED spotlight. The spotlight will have a position of {-5, 3, -5}, a cone direction of {1, -1, 1}, an inner cone ratio of 0.92, an outer cone ratio of 0.9, and a radius of 10.
* The spot light is the most complicated light you’ll be working on today, and the one you’ll be making has multiple types of attenuation going on.
* Fortunately, a spotlight is nothing but a directional point light, so your life can be made easier by starting with a working point light, and giving it a direction, inner, and outer cone ratios.

**Multi-pass Rendering**

* We’ll be doing multiple passes for our scene today to apply 3 different types of lighting to the same objects using 3 different light shaders. This means rendering the objects multiple times with different shaders. To get the lights to layer properly we’ll need to change some settings to get the multiple passes to render the way we want them to.
* First off the **depth function** of our **depth stencil description** will need to be changed. Normally we have this at **less than**, to filter out any pixels that are at the same depth or behind our foremost pixel. However with multiple passes we’ll be rendering the same object, at the same position, multiple times. Since we still want to see all of the passes we’ll need to change this to **less than or equal**.

**Blending**

* The last, and most important part of getting the lights to show up together, is blending.
* As with everything in DirectX 11, we’ll need to setup a description for it. The D3D11\_BLEND\_DESC is very powerful, but can be kind of intimidating at first look on MSDN.
* First things first, we’ll need to set **BlendEnable** to true, for obvious reasons.
* The **SrcBlend** and **SrcBlendAlpha** handle how we’ll treat blending with the pixel returned from the pixel shader. SrcBlend handles the RGB values, while SrcBlendAlpha handles the alpha values of the pixel. We can handle these separately for added control, but for what we’re doing today, we’ll be using a flag called **D3D11\_BLEND\_ONE**.
* D3D11\_BLEND\_ONE sets all our blend values to 1. This is the default for our SrcBlend values, because we normally want to see ALL of the pixel we’re creating.
* The next two values that need to be set in our blend description are **DestBlend** and **DestBlendAlpha**. Just like SrcBlend, these two handle color and alpha channels separately. However, instead of handling the blending values of the pixel we get from our pixel shader, it handles the blending values of the pixel that we’re looking trying to write to in our render target. Usually these are defaulted to D3D11\_BLEND\_ZERO because we want to see all of the pixel we’re creating in the pixel shader, and none of the pixel that was there before, get it? However because this is going to be used for multiple passes we want to see ALL of the pixel that is in our render target (the previous pass, let’s say the directional light in this case) and we want to see ALL of the pixel that we are going to get from our pixel shader (the next/current pass, in this case our point light). So instead of setting it to zero we want to set it to **D3D11\_BLEND\_ONE**.
* The final really important settings are **BlendOp** and **BlendOpAlpha**. These handle how our source (pixel shader) pixel and destination (render target) pixels are combined. When it comes to lighting we just want to **add** them together (think of it like if you turn on a light, you add light to the room). Because of this, we want to set them both to**D3D11\_BLEND\_OP\_ADD**.

**Blending Notes**

* This is how you would setup blending for lighting purposes. Let’s say you had an object that you wanted to just be transparent based off of its alpha value of the pixel in our pixel shader. If we set our SrcBlends to BLEND\_SRC\_ALPHA, and our DestBlends to BLEND\_INV\_SRC\_ALPHA and then add them, we’d end up with a transparent object. For example if our pixel shader pixel had an alpha of 0.3 (think of that like 30% visibility) we’d want our render target pixel to fill in the remaining 0.7 (We can see 70% of the render target through the pixel shader’s pixel). BLEND\_INV\_SRC\_ALPHA takes the 0.3 alpha value of the pixel shader and subtracts it from one. And as we all know, 1.0  – 0.3 = 0.7.

You can find more information about the blend description at:

<http://msdn.microsoft.com/en-us/library/windows/desktop/ff476200(v=vs.85).aspx>

More about the blend factors at:

<http://msdn.microsoft.com/en-us/library/windows/desktop/ff476086(v=vs.85).aspx>

And more about the blend operations at:

<http://msdn.microsoft.com/en-us/library/windows/desktop/ff476088(v=vs.85).aspx>