Assignment 5 – Models & Lights

Due by 11:59PM on Thursday, February 14th.

# Overview

Now that you can draw and control entities, and move around in the 3D world, it’s time to get more interesting geometry into your engines. You’ll be loading 3D models from .OBJ files and using them in place of, or in addition to, the geometry already in your scene. Since you can already draw sets of triangles, this just boils down to a new way of filling up your vertex and index buffers.

Once you’ve got 3D objects in your scene, you’ll quickly realize how bland they look without any sort of lighting or shading. For the second half of this assignment, you’ll be adding lighting to your shaders to accentuate the shape of your 3D models. You’ll need at least two directional lights for this assignment.

## .OBJ Files

As we discussed in class, .OBJ files are one of the easiest model files to handle because of their simplicity and the fact that they’re text-based. Included in the .ZIP file on MyCourses are several sample .OBJ files for use with this assignment and a text file with some OBJ loading code. You can read more about [the file format on Wikipedia](http://en.wikipedia.org/wiki/Wavefront_.obj_file), or refer to the lecture slides from class.

.OBJ files generally contain 3 useful pieces of information per vertex:

* Vertex position in 3D space
* UV coordinate (for texture mapping)
* Normal (mostly used for lighting)

.OBJ files also contain information about the faces (triangles) of the 3D model. The vertices of these faces are comprised of combinations of the above pieces of data. The files can get fancier than this, but your engine only needs to support the three basic pieces of information. That’s all you’ll need when you add lighting later in this assignment (and textures in the next assignment).

# Task Overview

Here is the high-level overview of tasks, which are explained in more detail on the pages that follow:

* **Load meshes** from external .OBJ files
* Update C++ and your shaders to support a **single diffuse directional light** on your objects
* Add **at least one more light** to your scene
* Ensure you have no **warnings**, **memory leaks** or **DX resource leaks**

# Task 1: Loading 3D Models

## Updating your Vertex Definition and Shaders

Before actually loading an OBJ file, you’ll need to make sure your vertex definition is compatible. Go into your Vertex.h file and adjust the Vertex struct as follows:

* Remove the XMFLOAT4 color
* Add an XMFLOAT3 normal
* Add an XMFLOAT2 uv

This will break your existing code for the time being. You’ll need to adjust *any and all* vertices you’ve created in previous assignments by removing the colors, then adding normals and uv coordinates. The normals can be (0,0,-1) to point at the camera, and the uv’s can be (0,0).

If you recall, the input to a vertex shader must match the layout of the vertex buffer. This means you’ll need to also update the input struct that’s defined in VertexShader.hlsl. Do this by *removing* the float4 color, and *adding* both a float3 normal (with the NORMAL semantic) and a float2 uv (with the TEXCOORD semantic). Remember order matters coming into the pipeline: make sure it matches your C++ Vertex struct.

Since you no longer have a color value coming into the vertex shader, you’ll need to change the VertexToPixel definitions in *both* shaders. Remove the color variable from both shaders’ VertexToPixel structs. You’ll also need to remove the line in the vertex shader that was copying the input color to the output color. Lastly, you can simply hardcode a color value to return in the pixel shader for now, like so: return float4(1,0,0,1); All your shapes would then be red, or whatever color you choose.

Test these changes **before moving on**! You should see all of the same shapes, but with a solid color.

## Updating the Mesh Class

Your existing Mesh class will be responsible for loading 3D models in addition to its existing tasks. Create a second constructor that accepts the name of a file to load (a char\* parameter). This will allow you to either pass in data for the mesh, or simply provide a filename to load. Inside this new constructor, use the provided model-loading code (see below) to read the file and store the data in several temporary std::vector objects. Then it’s up to you to use that data to create new Vertex and Index Buffers.

Inside the ZIP file from MyCourses, you’ll also notice a file called “Assignment 5 - OBJLoader.txt”. This file contains the C++ code you’ll need to copy/paste into your new Mesh constructor. This code works, but it is just a snippet (not an entire method). Copy and paste it into your new mesh constructor, adjust the *filename* variable and then utilize the “verts” and “indices” variables at the end of the code to create vertex and index buffers for this particular Mesh.

It may be useful to move your existing DirectX buffer creation code to a helper method that takes the same data as the original Mesh constructor. This way, both constructors can simply call this method (rather than duplicating that code a bunch of times). If you do this, use &verts[0] and &indices[0] to pass the address of the first element of the std::vectors to the method.

## Getting Models

Copy the provided example .OBJ files to a folder in your project. There are six models provided:

* Cone
* Cube
* Cylinder
* Hexlis
* Sphere
* Torus

You could drop them directly into the Debug folder, although it might be a little cleaner to instead create an “Assets” folder with sub-folders for “Models”, “Textures”, etc. in a higher folder. In either case, remember that the program’s “current working directory” will change depending on how you launch your program (through Visual Studio, or running the .exe manually). At a minimum, be sure your relative path is correct when running in Visual Studio.

Eventually, you might find it useful to write some code to automatically check both locations. However, that is not necessary for this assignment.

At this point, you should be able to test your updated Mesh class. Create some new meshes (and associated entities), or simply replace existing ones by using your new constructor overload, and run your program. **Don’t move on** until you can see some new geometry on your screen.

## Other Formats?

For those who want to think ahead, here is some info on other ways to import 3D models. None of these are required for this assignment.

[Here’s an article](http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/how-to-work-with-fbx-sdk-r3582) on using the FBX SDK to load mesh info from FBX files directly. The tutorial is quite helpful, as the FBX SDK is not as straightforward as you might expect.

You could also use the Open Asset Import Library found at <http://www.assimp.org/>

# Task 2: Adding Lights

Now that you have some new (and probably rather bland looking) shapes on your screen, it’s time to add lighting to your scene. You’ll be working in both C++ and HLSL for this. This assignment will guide you through the basics of adding a simple directional light source with both diffuse and ambient light. Then you’ll add some more.

## Lighting: C++ Side

C++ will be responsible for defining the actual data of the light(s) in your scene: the location, color and/or direction, depending on the types of light you implement. This data will need to be sent to each shader for each entity you intend to draw (in our basic, un-optimized engines anyway).

You’re going to define a basic “Light” struct and pass lights to each object right before it’s drawn.

An advanced engine might have a Light Manager or Renderer class in charge of passing the correct lights to various entities depending on distance and attenuation, as well as moving or updating the lights themselves.

Lights in an advanced engine might even have their own transformations for position and orientation, allowing the lights to move or rotate over time. They could inherit from a GameEntity class, or be a component added to GameEntities.

For now, define a struct called “DirectionalLight” in a new header file (Lights.h perhaps) with the following variables, *in this specific order*:

* XMFLOAT4 AmbientColor
* XMFLOAT4 DiffuseColor
* XMFLOAT3 Direction

Order matters here because we’re going to duplicate this struct in the Pixel Shader soon, which will then be used within a constant buffer. Vectors in constant buffers cannot cross 16-byte boundaries, so the 3-component vector will need to come last. This ensures the HLSL compiler doesn’t pad our struct without us knowing. An advantage of using the Simpler Shader class is that it allows us to essentially disregard that padding between individual variables in a constant buffer, but multiple variables inside a single struct could still be problematic.

Create a field of type DirectionalLight in Game.h (no pointer necessary), and then initialize it in the Init() function. You don’t need to call new since we’re not dealing with a pointer. Simply set the 3 variables with some default values:

* AmbientColor should be subtle. Example: (0.1, 0.1, 0.1, 1.0)
* DiffuseColor should be bright. Choose any color. Example: (0, 0, 1, 1)
* Direction can be any valid direction. You’ll be normalizing it in the shader. Example: (1, -1, 0)

Lastly, you’ll need to pass the entire light struct into the pixel shader before drawing each entity. You’ll be using the SetData() method for this, like so:

pixelShader->SetData(  
 “light”, // The name of the (eventual) variable in the shader  
 &light, // The address of the data to copy  
 sizeof(DirectionalLight)); // The size of the data to copy

Even though you haven’t added this variable to the shader yet, the code should still run without errors at this point. The *Set* methods of SimpleShader are designed to fail gracefully (and return false) rather than breaking if the variable isn’t found.

## Lighting: Vertex Shader

You should have a normal coming into the vertex shader at this point (from earlier in the assignment). The only thing you’ll really need to do at this point is apply the world transformations to the normal and pass it on to the pixel shader. Passing data to the pixel shader requires you to add an additional variable to the output structure (VertexToPixel).

Add a float3 called “normal” to the VertexToPixel struct, and use the semantic “NORMAL” for this variable. Order doesn’t matter here, as we only have to worry about padding and boundaries when dealing with data inside constant buffers (but not input/output structs).

Inside the shader function itself, you’ll need to copy the input struct’s “normal” to the output struct’s “normal”. However, if the geometry has some kind of transformation being applied by a world matrix, that matrix also needs to be applied to the normals. There are two issues however: Translation isn’t useful when transforming the normal, and the world matrix is 4x4. Both issues can be solved by casting the matrix to a 3x3 before applying it, like so:

output.normal = mul( input.normal, (float3x3)world );

Remember that this is only valid if you’re applying a uniform scale on your object: a scale with the same value on each axis. To properly transform a normal using a non-uniform scale, you’d need to use the inverse transpose of the world matrix. You’d calculate that matrix separately in C++ and pass it into the vertex shader as an extra variable in the constant buffer (but this is for the most part unnecessary for this assignment).

## Lighting: Pixel Shader

There are several additions you’ll need to make in the pixel shader: updating the VertexToPixel struct to be able to use the normal coming in from the vertex shader, creating a Light struct and then adding a constant buffer with a light variable (so C++ can actually pass data in). Then you’ll need to actually write the code to compare the surface’s direction to the light direction and calculate the final pixel color. We’ll tackle one step at a time:

**Updating the Input Struct**

Update the VertexToPixel struct so it includes a float3 normal (with the NORMAL semantic). This will allow the pixel shader to have access to the interpolated normal from the vertices of the triangle. Your first step in the shader function itself is to normalize this vector, as interpolation could result in non-unit vectors. Call the normalize function, passing in the input.normal variable, and overwrite the old value by storing it back in the same variable.

For quick testing, make the next line in the shader a return statement that returns the normal as a float4 vector, like so:

// Creates a float4 from a float3 and one more value  
return float4( input.normal, 1 ); // This line should be temporary

Run your program. You should see a mix of red, green, yellow and black colors (and potentially blue if you move the camera to the backside of the objects). These are the normals of the vertices being visualized as colors. Once you see correct normals, you can remove the above return statement.

**Defining a Light**

This step will require you to create two things: a struct, and a constant buffer. First, create a struct called *DirectionalLight* in the file (above of the main shader function). The variables in this struct should match the layout of your Light struct from C++, although the names can be different if you want (no semantics are necessary as this is not a shader input/output struct):

* float4 AmbientColor
* float4 DiffuseColor
* float3 Direction

Once you have the struct, you’ll need to declare a variable of that type which can be filled up with data from C++. This means you’ll be creating a constant buffer next. Remember that they’re defined with the cbuffer keyword, the identifier is arbitrary and they should be bound to a register (b0 is fine for this one). Refer to the Vertex Shader file or the lecture slides from class for the exact syntax.

Once you have your empty constant buffer, simply define a variable of type *DirectionalLight* named “light” inside the cbuffer. The variable’s name will be referenced from C++, so make sure it matches what you’ve done there already.

Test whether or not this data is making it to the pixel shader by simply returning the DiffuseColor of the light temporarily. Once you’re sure the data is getting there, move on to the next step.

**Implementing a Lighting Calculation**

You’ve got all the data necessary to calculate how the light is hitting this particular pixel. Here are the remaining calculations to get this done:

* Calculate the normalized direction *to the light*
  + Negate the light’s direction, normalize that and store in a float3 variable
  + You can’t store it back in the light variable because it’s a *constant*
* Calculate the light amount using the N dot L equation
  + Use the dot(v1, v2) function with the surface’s normal and the direction *to the light*
  + The normal should already be normalized from a previous step
  + The dot product result can be negative, which will be problematic if we have multiple lights. Use the saturate() function to clamp your result between 0 and 1
* Return the final surface color based on light amount, diffuse color and ambient color
  + Scale the light’s diffuse color by the light amount
  + Add the light’s ambient color
  + Return the result

Your program should now be applying your light to all of the objects being drawn with this shader. If the lighting seems off to you, be sure you’re properly negating the light’s direction. You can always temporarily return numbers as color values for quick and dirty visual debugging.

# Task 3: More Lights!

As a finishing touch, add **at least one more directional light** to your scene. This will require you to:

* Create a second light in C++ (ideally with a different color and direction)
* Pass that second light to the pixel shader
* Define a second light variable in your pixel shader’s constant buffer
* Redo the same lighting calculations as before (light direction, N dot L and final light color)
* Add the results of both lights together for the overall final color

In the shader, you can either hardcode these steps or create a helper method that accepts a normal and a light object, and returns a final light color. Then just call that method twice, adding the results.

# Odds & Ends

Here are some common additions or issues you may run into. None of these are required, but I often get asked about these topics:

## Arrays in Constant Buffers

It’s enticing to create an array of lights in your constant buffer, allowing you to passing in multiple lights easily. The potential problem is here again is data packing rules. The compiler will add padding inside the array to respect 16-byte boundaries. You can definitely use an array, but you’ll need to account for this padding on the C++ side, as the raw size of the array in the constant buffer may not match the size of the array in C++ if you don’t.

## Other Types of Lights

While not required for this assignment, you may be interested in implementing point lights and/or a specular light calculation. Use the slides from class as a reference, and/or e-mail me with any questions!

## All My Shapes are Red

Since we’ve removed the individual colors from the vertex definitions, all of your shapes are potentially the same color. If you’d like to be able to define a different color for each thing you draw, the material class seems like a good candidate to hold this information. Each material could then have a “surface color” field, similar to materials in engines like Unity.

To do this you’d need to update your material class in C++ to hold a color. Then create an extra float4 variable in the pixel shader’s constant buffer so you can use that color as part of the lighting equation (remember we multiply the light by the surface color). Lastly, ensure you’re sending color in during the PrepareMaterial() method.

# Deliverables

Submit a zip of the entire project to the appropriate dropbox on MyCourses.