Graphics – Work for Part 2, Week 8

Introduction

So far, you have been doing the lighting calculations on a per-vertex basis in the vertex shader. This has been implementing the Gouraud shading model where colours are calculated at the vertices of polygons and then we let DirectX interpolate the colours between the vertices to give us the colour for each pixel.

The first thing you will do this week is change your code to perform lighting calculations on a per-pixel basis in the pixel shader. This will enable you to implement specular highlighting to give a much more realistic effect.

I would suggest testing this out first on a node that draws one of the geometric shapes such as a teapot. Because the teapot contains a lot of curves, it really shows the impact of the different lighting models better than a cube. Also, doing it on a non-textured object makes it easier to see the impact of specular lighting effects.

How you handle the different types of light is up to you. You can just hard-code the values in the constant buffers or you might choose to implement a lighting collection in the DirectXFramework class and implement a class hierarchy for the different types of light. Then your node classes can retrieve the collection of lights when they are ready to render an object and populate the constant buffer accordingly.

As you can see, the further we go in this module, the less prescriptive I am being in the tutorials each week and leaving more of the design decisions up to you. If you have reached this point, this gives you a chance to show what you can do which will influence the grade you achieve in your final submission.

Using the Pixel Shader to Calculate the Lighting Colour

There are three things we need to do before we can move the calculations to the pixel shader. The following assumes we are not dealing with texture coordinates – if you are lighting a textured object, then you also need to factor in the texture coordinates (note that texture coordinates do not need any transformation in the vertex shader. They should get copied to the output structure):

1. We need to copy the normal and position in world space to the output from the vertex shader (which will be interpolated by DirectX and input to the pixel shader). The normal is needed for all of the lighting calculations and the position in world space will be needed when you calculate the vector back to the eye/camera positionYour vertex output structure might look something like the following:

```
struct VertexOut
{
        float4 OutputPosition    : SV_POSITION;
        float4 Normal            : TEXCOORD0;
        float4 WorldPosition     : TEXCOORD1;
};
```

Your vertex shader needs to perform the necessary calculations to populate these values.

Note the odd semantic names (TEXCOORD0 and TEXCOORD1) on the Normal and WorldPosition. HLSL requires semantic names on all fields that will be passed between shaders. However, there is no specific meaning attached to these two fields. So we can use any valid semantic name we want that is not reserved for a specific use. In this case, we use TEXCOORD0 and TEXCOORD1 even though neither of these is a texture coordinate. Just consider it one of the weird aspects of the HLSL language.

2. We need to add the eye position and the specular power to the constant buffer. I also find it can result in nice effects to have different colours for the diffuse reflection and the specular reflection. This is particularly true if you want to use a colour for the diffuse reflection but keep white for the specular highlights.

   The eye position should be the same as the one used by your camera. You can retrieve the current camera position from the camera object and put it into the constant buffer.

   One thing to be aware of is that your constant buffer should be a multiple of 16 bytes in size. This is not always needed, but can cause problems on some video cards if it is not. I would suggest calculating up the size of your constant buffer and adding some pad if it is not a multiple of 16. For example, if the size of your constant buffer is 12 bytes short of a multiple of 16 bytes, add a pad field at the end that looks something like (note that a Vector 3 contains 3 floats, each of which is 4 bytes long, giving a total of 12 bytes):

   ```
   Vector3        pad;
   ```

   And similarly in the shader cbuffer structure:

   ```
   float3         pad;
   ```

   This field is not used, but it does ensure that the buffer is a multiple of 16 bytes in size.

3. Your pixel shader will now need to access the constant buffer. So you need to add a line to your rendering code in your node classes to make the constant buffer visible to your pixel shader (if you did not do this already). The line to add is:

   ```
   _deviceContext->PSSetConstantBuffers( ,  , _constantBuffer.GetAddressOf());
   ```

   Put this line just after your call to VSSetConstantBuffers. Note that if, after you have moved all of your colour calculation to the pixel shader, your window shows as all black, it will be almost certain that you have missed this line out (I know because I forgot it initially when writing the examples for this week and it took me a while to figure out why my window was all black!).

Now you can do the lighting colour calculations in the pixel shader. Start by just doing the diffuse calculation to make sure this still works and then add in the specular highlights. You can use any

model you wish to calculate diffuse and specular shading, but put a comment in saying which one you are using and say why you chose this method in your implementation log.

Implementing Point and Spot Lights

Once you have directional lighting working, you might choose to implement point and spot lights. These will require additional fields to be added to the constant buffer (such as the a, b and c values for the attenuation for point and spot lights and the cosines of the inner and outer angles for spot lights).

A couple of extra HLSL functions that will be useful are:

length(x)	Calculates the length of vector x.
smoothstep(min, max, x)	Performs the smoothstep calculation.

For these lights, you will need to pass in the position of the light in the constant buffer rather than the vector from the light. You will then need to calculate the vector back to the light using the position of the light and the world position. The distance to the light is the length of the vector back to the light.

Blending Terrain Textures

The next step in this tutorial now look at how you can blend textures to give a more realistic terrain.

Using this approach, we will apply multiple textures to each square in the grid and use a blend map to determine how much of each texture is displayed.

To do this, we first need to change the vertex format since we will need two sets of texture coordinates – one to index into the blend map and one to index into the individual textures. The vertex format I used (and is expected by the shader I have supplied) is:

```
struct TerrainVertex
{
	Vector3 Position;
	Vector3 Normal;
	Vector2 TexCoord;
	Vector2 BlendMapTexCoord;
};
```

This structure, the modified vertex layout description required and the two additional methods described below are provided in BlendedTerrain.c. The shaders that handle this are supplied in TerrainShader.hlsl.

You calculated the TexCoord values when creating the terrain grid two weeks ago. You now need to add the BlendMapTexCoords. When creating the vertices, the BlendMapTexCoord values for U and V should be the positions into the blend map which is stretched across the entire grid

The tiled textures are loaded using the method LoadTerrainTextures which I have supplied for you. This loads the five different textures into a Texture2DArray element. Once again, this is code that is not very well documented in the official documentation or in books, so I have supplied it to make things easier for you.

I have provided a number of example textures that you can play with. The code I have provided expects these to be in a folder called Terrain that is added to your Visual Studio project folder (do not add them to the Visual Studio solution though).

You will see that the textures are .dds files. This is a format used by DirextX for some textures. The code I have provided uses the function CreateDDSTextureFromFileEx to load the textures. This function is provided by the files DDSTextureLoader.h and DDSTextureLoader.cpp. These come from the DirectX Toolkit. I have provided them for you and you should add them to your solution.

Now you need to create the blend map. This is supplied to the shader as another texture. You could create it using an art package using different values for r, g, b and a at each point to indicate how much of each of the overlaid textures are displayed. However, this will not work for procedurally generated terrain. Instead, I have provided the starting point for code that will generate a blend map at run time. This is provided in the method GenerateBlendMap. You will need to add code to calculate appropriate values for each position in the array. A value of 0 for r, g, b and a at each position will mean that only the grass texture is used over the entire terrain.

If you use the textures as loaded in the example I have provided, the value in the R component determines how much of the dark dirt texture is blended in. The value in the G component determines how much of the stone texture is blended in. The value in the B component determines how much of the light dirt texture is blended in. Finally, the value in the A component determines how much of the snow component is blended in. Each component has a value between 0 and 255. You should use the height values at each point in the grid and possibly the amount of slope to determine which textures to use – how you do this is entirely up to you based on the type of terrain you want to create.

Once the R, G, B and A values have been calculated, they are combined into one 32-bit value where the A component occupies the top 8 bits, the B component the next 8 bits, the G component the next 8 bits and the R component the lowest 8 bits.

Once you have the blend map and the terrain array, these need to be passed to the shader before the terrain is rendered. The lines of code needed to do this are:

```
_deviceContext->PSSetShaderResources(0, 1, _blendMapResourceView.GetAddressOf());
_deviceContext->PSSetShaderResources(1, 1, _texturesResourceView.GetAddressOf());
```

Don’t forget to make the changes to the input assembler layout and to the code that compiles the shaders to take account of the new shader file name.

When you run your code, the result will depend on the values you put in your blend map.