Graphics . Tutorial for Part 2 Week 3

Exercise 1

Your starting point for the exercise this week is the solution found in the file DirectX_Cube_Week3. This contains a rotating cube that is just illuminated by ambient light. Your task this week is to add a directional light source and implement directional lighting on the cube. This exercise will be the first time that you have had to decide on your own edits for the shader code (in this case, shader.hlsl).

Note. The shaders are not compiled when the rest of the project is built. They are compiled at run-time. Therefore, if you have any compiler errors in your shader code, they will not show up until run-time. An exception will be thrown if errors occurred while compiling the shaders. Make sure you read the message in the error box that is displayed carefully to identify the errors in your shader code.

The starting solution contains a few key changes to the solution you saw last week:

- There are now four vertices for each face of the cube. The reason for this is that we need to calculate separate normals for each vertex and it is also in preparation for when we apply a texture to the cube. So you will see that the vertices array in Geometry.h is larger. This change has also resulted in the list of indices changing,
- The position of each vertex is now represented as a Vector3 rather than a Vector4. Since the W co-ordinate will always be 1 in the source vertices, there really is no need to provide it with each vertex. We just set it to 1 in the vertex shader before multiplying the position by the transformation matrices as seen in the line below:

  ```
  vout.OutputPosition = mul(worldViewProjection, float4(vin.InputPosition, 1.0f));
  ```

  Note that in the shader file, the format of the vertex has now changed to float3 to match the format of the vertex in the C++ code. Also note in the vertex shader how the position is expanded out to a float4. Finally, to support this change, note that the format of the vertex position in the input array, vertexDesc in Geometry.h has been changed to reflect this change.
- A material colour (that is, the base colour of the cube) and an ambient colour field has been added to the constant buffer in both geometry.h and in the shader file. The value of these fields are set before the cube is rendered in the C++ code. The default colour of the cube has been set to white (Vector4(1.0f, 1.0f, 1.0f, 1.0f)) and the ambient light colour has been set to grey (Vector4(0.5f, 0.5f, 0.5f, 1.0f)). Right now, the shader just multiplies the material colour by the ambient light colour and stores the result in the colour field for each output vertex using the line below:

  ```
          .       =                          *
  ```

Using this as a starting point, you now need to add code to:

a) Generate normals for every vertex in the model
b) Expand the constant buffer to pass a directional light source through to the shader and add code to the vertex shader to calculate the impact of the directional light on the colour of the vertex (we will be doing gouraud shading here and just calculating the colour at the vertices, letting Direct3D interpolate the colours between the vertices. We will get to per-pixel lighting later).

The stages you need to go through are as follows:

Stage 1

The first thing you will need are normals for each of the vertices. In practice, most 3D modelling products will create normal for a model. However, there are times when this cannot be done (for example, for procedurally generated terrain). So in this exercise you will write the C++ code to calculate the vertex normals.

This stage adds the extra fields needed to the vertex structures:

- Add a Vector3 field for the normal to the Vertex type in Geometry.h and add an additional Vector3 to each entry in the vertices array. For now, set the x, y and z components for the normal to 0 – you will be adding code to calculate the values for the normal in stage 2.
- Update the vertexDesc array in Geometry.h to add information about the additional field for the normal. You should be able to work out what the extra line should be by studying the line for the position (to give you the type) and by looking at a solution to the exercise from last week to see how the colour was added to this array. The semantic name you should use for the extra line is “NORMAL”.
- Finally, since the input vertex structure used in the shader should always match the one in the C++ code, the normal should be added to the input vertex structure in the shader, i.e. it should become:

```
struct VertexIn
{
	float3 InputPosition : POSITION;
	float3 Normal        : NORMAL;
};
```

Stage 2

Now, before we create the vertex and index buffers, we need to generate the vertex normals. To do this, first we need to calculate the normal for each polygon. Then, for each vertex that is part of that polygon, we need to add the polygon normal to each vertex normal and add one to a count of polygons that have contributed to that vertex normal. Finally, we divide each vertex normal by the number of polygons that the vertex is part of. The nice thing about this is that it takes into consideration the lesser contribution from smaller polygons. This is because the length (or magnitude) of the normal of each polygon is proportional to the area of the triangle it was created from since it was created by taking the cross-product of the two vectors that span the polygons edges.

Firstly, you need to create an array the same size of the vertices array that will store the contributing counts (i.e. stores how many polygons the vertex is part of). Each element in the array will just be an integer.

The logic to create the vertex normal is then as follows. This logic assumes that each vertex normal starts as (0, 0, 0) – as you did in stage 1.

```
For each Vertex, set the corresponding contributing count array entry to 0.
For each Polygon
      Calculate the normal for the polygon (see the algorithm below)
      Add the normal for the polygon to the vertex normal for each of the
      3  vertices for that polygon and add 1 to the contributing count for
      each of the vertices
Once all the polygons are processed,
     For each vertex
        Divide the summed vertex normals by the number of times they were
              contributed to
        Normalize the resulting normal vector
```

To calculate the normal for a polygon, do the following:

```
  Get the 3 indices of the vertices that make up the polygon
  Get the 3 vertices for those indices
    Construct vector a by subtracting vertex 0 from vertex 1.
    Construct vector b by subtracting vertex 0 from vertex 2.
    Calculate the normal vector from vector a and b using a x b (the cross
         product of a and b)
```

To do all of this, you will need to use methods and properties of the Vector3 class which you can find information on at https://github.com/microsoft/DirectXTK/wiki/Vector3. Note that you do not need to write your own code to calculate the cross product.

For example, to calulate the cross-product of vectors a and b, you would use:

Vector3 crossproduct = a.Cross(b);

Note. Make sure you do all of this before you create the Direct3D vertex and index buffers.

Stage 3

Now that we have calculated the normals, we need to update the rendering stage. First, we add additional fields to the constant buffer. There are two key things we need to add:

- We will need to multiply each of the normal vectors by the world transformation so that they are adjusted for the position of the object in world space. So we need to add the world transformation matrix to the constant buffer.
- We need to add the directional light vector and colour to the constant buffer.

Change the constant buffer in Geometry.h so that it looks like the following:

```
struct CBuffer
{
      Matrix      WorldViewProjection;
      Matrix      World;
      Vector4     MaterialColour;
      Vector4     AmbientLightColour;
      Vector4     DirectionalLightColour;
      Vector4     DirectionalLightVector;
};
```

You also need to update the constant buffer in the shader so that it has the same structure.

Stage 4

You now need to add code in the vertex shader in shader.hlsl to:

- Multiply the world transformation matrix by the normal to give an adjusted normal (which will be a float4).

- Take the dot product of the adjusted normal and the vector back to the light source (note that the vector being passed in the constant buffer is the vector from the light source) and normalise it. Then make sure it is a value between 0 and 1. This will be the amount of diffuse light hitting the vertex (a float).
- Finally multiply the amount of diffuse light by the directional light colour and add the ambient light colour to give the total amount of light on the vertex. Ensure that each component is a value between 0 and 1. Then multiply by the material colour for the object. This is the final colour for the vertex.

You are likely to find the following additional HLSL functions useful to do this:

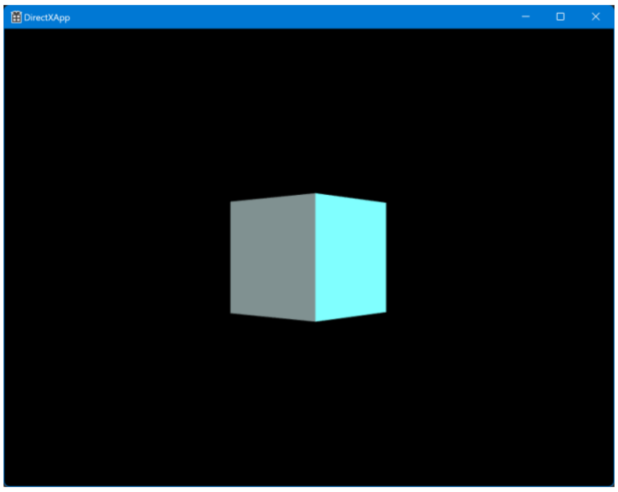
dot(a, b)	Calculate the dot product of vectors a and b
normalize(v)	Normalise the vector v.
saturate(f)	Clamp the value f so that it is in the range 0 to 1. If used on a float4, each component will be clamped to a value between 0 and 1.

Stage 5

Finally, you want to add code to the Render method in DirectXApp.cpp to add the additional information to the constant buffer before it is passed to the shader. You need to store the world transformation matrix, the colour of the directional light and the vector of the directional light. To start with, I would suggest using the following values for the directional light:

```
constantBuffer.DirectionalLightVector = Vector4(-1.0f, -1.0f, 1.0f, 0.0f);
constantBuffer.DirectionalLightColour = Vector4(Colors::Cyan);
```

This will result in a cyan light that is shining from the top right and into the scene. This will result in the following sort of image as the cube rotates:



Exercise 2

At this point, you are probably bored with working with a cube. So let's look at other shapes.

Make a copy of your solution to exercise 1 as a starting point. Then, on Course Resources, you will find a zip file called GeometricObjects.zip. Unzip this file and copy the three files into the folder containing the source files for your project. Add all three files to your project in Visual Studio.

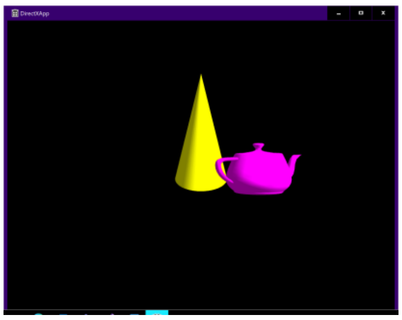
If you include GeometicObject.h into your main program, you will have access to a set of functions that will produce the vertices and indices for a number of different objects, namely:

- Box
- Sphere
- Cylinder
- Cone
- Teapot

All of the functions are commented in GeometricObject.h so you should read that header file carefully. There are a few things to note:

- These functions populate vectors that contain the vertices and indices for the objects. So you will need to create vectors of the required types (see the function prototypes) and pass these into the functions. You should also delete the vertices and indices arrays from geometry.h.
- The vertex structure used is defined in GeometicObject.h as ObjectVertexStruct. Although this is the same format as the VERTEX struct that you used in exercise 1, for completeness you should also delete the VERTEX struct from geometry.h and use ObjectVertexStruct instead.
- You should call one of the functions from your Initialise method and then calculate the normals for the object as you did in exercise 1.
- You will need to change the BuildGeometryBuffers function since your vertices and indices are now in vectors rather than arrays. The changes are minor, but you do need to be careful with them. In particular, one thing to keep in mind is that you should use the address of the first element of the vector as the starting address of the vectors of vertices and indices.

The following image shows a teapot and a cone generated by the provided functions. In each case, I used different material colours and used a white directional light.


DirectXApp