Objectives

1. To develop a basic understanding of the point light illumination model
2. To understand how to model light sources and object materials.
3. To understand the differences between per-vertex lighting and per-pixel lighting
4. To learn how to pass through data between different rendering stages of graphics pipeline
5. To understand how to implement ambient, diffuse, and specular light in shaders

All the exercises shown below are described based on Tutorial04.cpp, but they can be done directly based on what you have achieved in Lab 1 & 2. As normal vector is required for doing lighting, a normal vector must be provided for each vertex of your model. You may need to modify the original layout of the vertex data to pass the information of normal vectors to the vertex shader. In addition, if you would like to implement the light effect based on the original cube model, you may also need to edit the vertex list so that it fully provides all the vertices of the 12 triangles. This is because the required normal vectors are specified as an attribute of cube vertices. You can have a look at the sample code provided in Tutorial06.cpp to get a basic understanding of how to specify the normal vectors for your object.

Exercise 1. Per-vertex diffuse lighting

1. To pass light position to vertex shader, add a new element “lightPos” in the struct ConstantBuffer.

```
struct ConstantBuffer
{
        … …
        XMMATRIX mProjection;
        XMVECTOR lightPos;
};
```

2. Initialize and specify a proper value for the vector **`lightPos`** in the C++ code.
3. Open the vertex shader and add a float4 type element to the cbuffer structure ConstantBuffe. The order of the elements in struct ConstantBuffe must be identical to the order of the elements in ConstantBuffer you specified in your C++ code.
4. Add a new member in the struct SimpleVertex corresponding to vertex normal, say

```
struct SimpleVertex
{
    XMFLOAT3 Pos;
    XMFLOAT4 Color;
    XMFLOAT3 Normal;
};
```

5. Add a new element in the input vertex element layout corresponding to the normal vector associated to each vertex:

```
D3D11_INPUT_ELEMENT_DESC layout[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 28, D3D11_INPUT_PER_VERTEX_DATA, 0 },
};
```

6. Specify the vertices of the 12 triangles used for defining the cube:

```
SimpleVertex vertices[] =
{
    { XMFLOAT3(-1.0f, 1.0f, -1.0f), XMFLOAT4(0.0f, 0.0f, 1.0f, 1.0f), XMFLOAT3(0.0f, 1.0f, 0.0f)},
    { XMFLOAT3(1.0f, 1.0f, -1.0f),  XMFLOAT4(0.0f, 0.0f, 1.0f, 1.0f), XMFLOAT3(0.0f, 1.0f, 0.0f)},
    { XMFLOAT3(1.0f, 1.0f, 1.0f),   XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f), XMFLOAT3(0.0f, 1.0f, 0.0f)},
    { XMFLOAT3(-1.0f, 1.0f, 1.0f),  XMFLOAT4(0.0f, 0.0f, 1.0f, 1.0f), XMFLOAT3(0.0f, 1.0f, 0.0f)},

    … …
```

7. Modify the indices of the triangle list to be drawn corresponding to the new vertex list.
8. Open the vertex shader and specify the vertex shader input. Each argument in the input list must be followed with a semantic you defined in the structure "D3D11_INPUT_ELEMENT_DESC layout[]". For instance, for the layout described in step 5, the input of your vertex shader should be like

```
VS_OUTPUT VS( float4 Pos : POSITION,  float4 Color : COLOR, float3 N : NORMAL)
```

9. To render the cube properly, you may need to create the depth-stencil view.
   1). Add global variables relating to the creation of the depth-stencil and its view:

```
ID3D11Texture2D*        g_pDepthStencil = nullptr;
ID3D11DepthStencilView* g_pDepthStencilView = nullptr;
```

   2). To create the depth-stencil view, you need first create a depth stencil texture object

```
D3D11_TEXTURE2D_DESC descDepth;
ZeroMemory(&descDepth, sizeof(descDepth));
descDepth.Width = width;
… …
descDepth.BindFlags = D3D11_BIND_DEPTH_STENCIL;
… …
hr = g_pd3dDevice->CreateTexture2D(&descDepth, nullptr,
                                   &g_pDepthStencil);
```

   3). Create the depth-stencil view using the depth-stencil texture created above:

```
D3D11_DEPTH_STENCIL_VIEW_DESC descDSV;
ZeroMemory(&descDSV, sizeof(descDSV));
descDSV.Format = descDepth.Format;
descDSV.ViewDimension = D3D11_DSV_DIMENSION_TEXTURE2D;
descDSV.Texture2D.MipSlice = 0;
hr = g_pd3dDevice->CreateDepthStencilView(g_pDepthStencil, &descDSV,
                                  &g_pDepthStencilView);

… …
```

   4). Set render target using the created depth-stencil view

```
g_pImmediateContext->OMSetRenderTargets(1, &g_pRenderTargetView,
g_pDepthStencilView);
```

   5). In the Render( ) method, clear the depth-stencil buffer using the created depth-stencil view

```
g_pImmediateContext->ClearDepthStencilView(g_pDepthStencilView,
D3D11_CLEAR_DEPTH, 1.0f, 0);
```

10. In the vertex shader, apply the point-light illuminate equation to calculate light colour at each vertex. For instance,

```
float4 materialAmb = float4(0.1, 0.2, 0.2, 1.0);
float4 materialDiff = float4(0.9, 0.7, 1.0, 1.0);
float4 lightCol = float4(1.0, 0.6, 0.8, 1.0);
float3 lightDir = normalize(lightPos.xyz - Pos.xyz );
float3 normal = normalize(N);
float diff = max(0.0, dot(lightDir, normal));
output.Color = (materialAmb + diff* materialDiff) *lightCol;
```

Note that if you define `materialAmb, materialDiff, lightCol` outside the body of the vertex shader, they have to be declared as static variables.

11. Modify light position, material colour and light colour and observe how the cube is being illuminated.
12. Modify the material reflection coefficients so that only red light is reflected.

Exercise 2. Per-pixel diffuse lighting

1. Pass the required elements involved in the light equation to pixel shader, so that the quantities such as light directions and normal vectors are available for pixel shader. There are different ways of doing this. One way is to pass vertex position to rasterization stage and let the rasterization operation to create fragment positions and send them to the pixel shader.

```
struct VS_OUTPUT
{
    float4 Pos       : SV_POSITION;
    float3 Norm      : TEXCOORD0;
    float3 PosWorld  : TEXCOORD1;
};
```

2. Open vertex shader, specify the values for variables Norm and PosWorld.

```
VS_OUTPUT VS( float4 Pos : POSITION, float4 N : NORMAL)
{
    VS_OUTPUT output = (VS_OUTPUT)0;
    … …
    … …
    output.PosWorld = Pos.xyz;
    output.Norm = N.xyz;

    return output;
}
```

3. Open pixel shader and calculate the pixel colour in the same way as that described in step 10 in Exercise 1.

Exercise 3. Per-pixel specular lighting

1. To implement specular light, view direction needs to be used. You need first pass the camera's position to pixel shader.
2. Calculate reflected light direction **R** using HLSL function reflect( ).
3. Calculate the view direction **V**.
4. Normalize both **R** and **V** into unit vectors and then find the dot-product between **R** and **V**.
5. Find the specular value:

   float spec = `pow( max(0.0, dot(V,R)), f);`

   where f is a value in [1, 200] in general, a parameter used to specify the degree of shininess.
6. Calculate the specular colour using the equation

   SpecColour = Material_{spec} * `lightCol*spec;`
7. Add it with the diffuse component you have obtained in Exercise 2.

Exercise 4. Multiple materials and light sources

Draw three cube objects with three different surface material properties. Illuminate these cubes with two light sources: one is a static white light, the other is a red light rotating dynamically by the y-axis.