Objectives

1. To understand what environment mapping is.
2. To understand how to implement cube mapping effect using a cube map.
3. To learn how to create a shinny reflective object that reflects the environment.
4. To learn how to create a translucent refractive object in the scene.
5. To learn how to do billboarding in Direct3D.
6. To implement sprite-based particle system for fire and smoke animation.

All the exercises described below are based on D3D 11 tutorial 7, but they can also be done directly based on what you have achieved in the lab on lighting as well.

Exercise 1. Draw environment

The first thing you need to do is to create a cube map representing the environment. The process is the same as the creation of an ordinary texture object. The only difference is that an array of six textures is to be used.

1. Save the cubemap texture Skymap.dds provided for this lab to your VS 2019 project directory, or create a cube texture of your own using the texassemble texture tool which can be downloaded from https://github.com/Microsoft/DirectXTex. You can find some descriptions about its use from https://github.com/Microsoft/DirectXTex/wiki/Texassemble
2. Create a shader resource view and a cubemap sampler state for the cube texture:
```
ID3D11ShaderResourceView*          sky_TextureRV = nullptr;
ID3D11SamplerState*                sky_Sampler = nullptr;
```
3. Load a cube map to create a cube texture object using the CreateDDSTextureFromFile( ) method.
```
hr = CreateDDSTextureFromFile(g_pd3dDevice, L"Skymap.dds", nullptr,
                                          &sky_TextureRV);
```
4. Next, set the texture as a pixel shade resource in the Render( ) method:
```
g_pImmediateContext->PSSetShaderResources(1, 1, &sky_TextureRV);
```
5. Now create a sampler state to specify how the texture is to be used
```
D3D11_SAMPLER_DESC sampDesc;
ZeroMemory(&sampDesc, sizeof(sampDesc));
sampDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
sampDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
sampDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
sampDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
sampDesc.ComparisonFunc = D3D11_COMPARISON_NEVER;

hr = g_pd3dDevice->CreateSamplerState(&sampDesc, &sky_Sampler);
```
6. In the Render( ) method, specify the sampler state to be used:
```
g_pImmediateContext->PSSetSamplers(1, 1, &sky_Sampler););
```
7. Pass eye position to the vertex shader through constant buffer and make sure this eye position is the same as the one used in the function `XMMatrixLookAtLH( Eye, At, Up )` `for creating the view matrix.`

8. Create a vertex shader and a pixel shader objects for rendering the sky

```
ID3D11VertexShader*      g_pVertexShaderSky = nullptr;
ID3D11PixelShader*       g_pPixelShaderSky = nullptr;
```

   a. Write the vertex shader: VS_Environment( )
      A cube texture is sampled using view direction, which needs to be calculated in vertex shader and passed to the pixel shader to sampler cubemap colours:

```
struct PS_INPUT
{
   float4 Pos : SV_POSITION;
   float3 viewDir: TEXCOORD0;
};
```

```
PS_INPUT_env VS_Environment(VS_INPUT input)
{
    PS_INPUT_env output = (PS_INPUT)0;

    float4 inPos = input.Pos;

    output.viewDir = inPos.xyz;

    //Make sure the vertex position is always in front of the eye:
    inPos.xyz += eyePos;

    inPos = mul(inPos, View);

    inPos = mul(inPos, Projection);
    output.Pos = inPos;

    return output;
}
```

   b. Now write the pixel shader for sampling colour from an environment map:

```
TextureCube txSkyColor : register(t1);

SamplerState txSkySampler : register(s1);


float4 PS_Cubemap( PS_INPUT_Cubemap input) : SV_Target
{

        txSkyColor.Sample(txSkySampler, input.viewDir);

}
```

   c. Create vertex shader and a pixel shader objects for rendering the sky using the shaders written in step 8 and 9:

```
… …
hr = CompileShaderFromFile(L"Tutorial07.fx", "VS_Cubemap", "vs_4_0", &pVSBlob);
… …
hr = CompileShaderFromFile(L"Tutorial07.fx", "PS_Cubemap", "ps_4_0", &pPSBlob);
```

9. As a small geometric model is used to represent the environment and viewed from inside, depth-Stencil state needs to be properly configured to make environment be drawn properly.

```
ID3D11DepthStencilState* g_pDepthStencilStateSky= nullptr;
ID3D11DepthStencilState* g_pDepthStencilStateCube= nullptr;
```

Disable depth test when drawing the sky and enable the depth test when drawing an object in the scene:

```
 D3D11_DEPTH_STENCIL_DESC dsDesc;

 // Depth test parameters
 dsDesc.DepthEnable = false;
 dsDesc.DepthWriteMask = D3D11_DEPTH_WRITE_MASK_ALL;
 dsDesc.DepthFunc = D3D11_COMPARISON_LESS;

 // Stencil test parameters
 dsDesc.StencilEnable = false;
 dsDesc.StencilReadMask = 0xFF;
 dsDesc.StencilWriteMask = 0xFF;


g_pd3dDevice -> CreateDepthStencilState(&dsDesc,&g_pDepthStencilStateSky);


dsDesc.DepthEnable = true;
g_pd3dDevice->CreateDepthStencilState(&dsDesc, &g_pDepthStencilStateCube);
```

10. Create rasterization state. When a small cube is used as sky geometry, it is viewed from inside the cube. Thus the cull mode should set to be none or front face:

```
ID3D11RasterizerState* g_pRasterSateSky = nullptr;

 //create rasterize state
 D3D11_RASTERIZER_DESC rasterDesc;
 rasterDesc.CullMode = D3D11_CULL_NONE;
 rasterDesc.FillMode = D3D11_FILL_SOLID;

 rasterDesc.ScissorEnable = false;
 rasterDesc.DepthBias = 0;
 rasterDesc.DepthBiasClamp = 0.0f;
 rasterDesc.DepthClipEnable = false;
 rasterDesc.MultisampleEnable = false;
 rasterDesc.SlopeScaledDepthBias = 0.0f;

 hr = g_pd3dDevice->CreateRasterizerState(&rasterDesc, &g_pRasterSateSky);

 … …
 hr = g_pd3dDevice->CreateRasterizerState(&rasterDesc1, &g_pRasterSateCube);
```

11. Bind Depth-Stencil state, rasterization state, shaders, shader resoruce and sampler at the OM Stage:

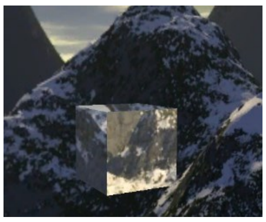
```
// Render the sky:
g_pImmediateContext->OMSetDepthStencilState(g_pDepthStencilStateSky, 1);
g_pImmediateContext->RSSetState(g_pRasterSateSky);
g_pImmediateContext->PSSetSamplers(1, 1, &sky_Sampler);
g_pImmediateContext->PSSetShaderResources(1, 1, &sky_TextureRV);
… …
… …
g_pImmediateContext->DrawIndexed( 36, 0, 0 );

// Render the cube:
g_pImmediateContext->OMSetDepthStencilState(g_pDepthStencilStateCube, 1);
… …
… …
g_pImmediateContext->DrawIndexed( 36, 0, 0 );
```

Exercise 2. Draw reflective shiny object

1. Write the vertex shader to draw the cube. To make the cube reflect the environment, both view direction and cube vertex normal are required to be passed to pixel shader.
2. Write pixel shader to render the cube, using the reflected view direction to sample the cubemap colour:

```
float4 PS(PS_INPUT input) : SV_Target
{
        … …
        float3 viewDir = reflect(-input.viewDir, input.Norm);
        return txSkyColor.Sample(txSkySampler, viewDir);
}
```

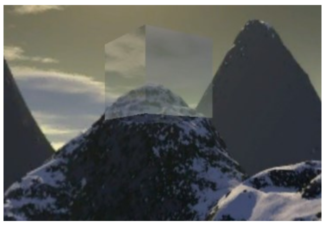
3. Draw another cube using the new render states and new shaders. Make sure that the depth stencil testing is to enabled. You may also need to create and configure the rasterization state for the cube in the scene so that only the front face of the cube is drawn



Exercise 3. Draw transparent shiny object

Edit the pixel shader you created in Exercise 2, using both the reflected and refracted view direction to sample the cubemap colour:

```
float4 PS(PS_INPUT input) : SV_Target
{
    … …
    float3 viewDir = reflect(-input.viewDir, input.Norm);
            float3 refrDir = refract(-viewDir, input.Norm, 0.9);

            float4 skyColor = txSkyColor.Sample(txSkySampler, viewDir);
            float4 refrColor = txSkyColor.Sample(txWoodSampler, refrDir);
    return   0.5*skyColor+  0.5* refrColor;

}
```



Exercise 4. Particle system-based fire animation

1. Create a list of quads of size [-1, 1]x[-1, 1]x[0, 1].
2. Create a blend state to enable colour blending.

3. Create depth-stencil state to disable depth testing.
4. Create rasterization state to make both faces of a quad visible.
5. Edit vertex shader
    a. to make each quad always face the view direction.
    b. to model the behaviour of each individual particle by writing a set of functions of time regarding particle’s position and colour.
6. Edit pixel shader to render the quad.
7. Fine-tune the particle parameters and the set of functions of time you created to animate fire and smoke.