Texturing in DirectX 11

Before attempting this tutorial make sure you have finished all previous tutorials, if you have not yet completed these tutorials then ask your tutor for help. In particular make sure that you have the previous lighting exercises finished, as these will be needed when combining your final pixel values.

# Texturing

Our scene is getting a little more interesting, but real-world objects are not typically uniformly coloured. What our objects lack is detail and texture. Texture mapping is a technique that allows us to map image data onto a triangle, thereby enabling us to increase the detail and realism of our scene significantly.

Direct3D uses a texture coordinate system that consists of a u- axis that runs horizontally to the image and a v-axis that runs vertically to the image, this is illustrated in Figure 1 below. The coordinates (u, v) identify an element on the texture called a texel. Also, notice the normalized coordinate interval, [0, 1], which is used because it gives Direct3D a dimension-independent range with which to work.

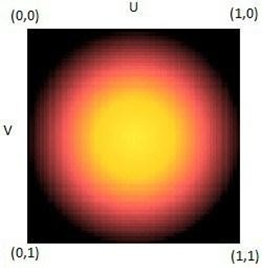


Figure 1: The DirectX texture UV coordinate system

For each 3D triangle, we want to define a corresponding triangle on the texture that is to be mapped to the 3D triangle. To do this, we modify our vertex structure once again and add a pair of texture coordinates that identifies a vertex on the texture.

## Step 1: Altering the vertex structure

Alter your vertex structure and vertex description to include texture coordinates as follows, your structure may look different to this. All you need to add is the sections related to the texture co-ordinates:

struct SimpleVertex

{

XMFLOAT3 Pos;

XMFLOAT3 Normal;

XMFLOAT2 TexC;

};

D3D11\_INPUT\_ELEMENT\_DESC layout[] =

{

{ "POSITION", 0, DXGI\_FORMAT\_R32G32B32\_FLOAT, 0, 0, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

{ "NORMAL", 0, DXGI\_FORMAT\_R32G32B32\_FLOAT, 0, 12, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

{ "TEXCOORD", 0, DXGI\_FORMAT\_R32G32\_FLOAT, 0, 24, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

} ;

More information on the input layout structure can be found [here](http://msdn.microsoft.com/en-us/library/windows/desktop/ff476180(v=vs.85).aspx), and further information about the whole input assembler stage of the pipeline can be found [here](http://msdn.microsoft.com/en-us/library/windows/desktop/bb205117(v=vs.85).aspx).

So, now every 3D triangle defined by three vertices also defines a 2D triangle in texture space (i.e. we have associated a 2D texture triangle for every 3D triangle)

This tutorial will discuss the texturing of a cube; however you are welcome to texture your own models instead.

# Step 2: Alter vertex data

Add a new XMFLOAT2 to each vertex in your vertex array. This should contain the texture coordinates for the vertex, you will need to think of appropriate co-ordinates and then input these into the vertex buffer along with all your other vertex data. These coordinates will be interpolated in the graphics pipeline down to the pixel level. Note: You may need to add extra vertices as a single texture coordinate will not work for multiple facets.

# Step 3: Adding texturing to our shader

Textures are assigned to a separate buffer inside our shader architecture, they are not included in the constant buffer with the other global variables we use. Also, we need to define a SamplerState to tell DirectX how to transform the texture data to fit into the correct size onscreen. Add the following code to the top of your shader file:

Texture2D txDiffuse : register( t0 );

SamplerState samLinear : register( s0 );

This gives us access to texture register 0, and also to sampler register 0. To now use this in your pixel shader you would have the following HLSL to get the pixel RGBA value at the location defined by the 2D coordinates in input.Tex:

float4 textureColour = txDiffuse.Sample(samLinear, input.Tex);

We now need to pass texture coordinates into the vertex shader, and then from the vertex shader to the pixel shader for use. You will need to add the following code to your shader file:

1. Add float2 Tex : TEXCOORD0; to vertex shader input and output, and the pixel shader input. E.g:

struct VS\_INPUT

{

float4 Pos : POSITION;

float2 Tex : TEXCOORD0;

};

struct PS\_INPUT

{

float4 Pos : SV\_POSITION;

float2 Tex : TEXCOORD0;

};

1. Pass the texture value inputted to the vertex shader into the output structure:

output.Tex = input.Tex;

1. Now ensure that the inputted texture coordinates are used inside the pixel shader.

# Loading a Texture

Before we can use the texture inside our shader code we need to read from an image file stored on disk and loaded into an ID3D11Texture2D object.

However, texture resources are not bound directly to the rendering pipeline; instead, you create a shader resource view (ID3D11ShaderResourceView) to the texture, and then bind the view to the pipeline. So two steps need to be taken:

1. Create the ID3D11Texture2D object from an image file stored on disk.

2. Call ID3D11Device::CreateShaderResourceView to create the corresponding shader resource view to the texture.

However, DirectX11 does not include functionality to load textures from disk. Therefore we are going to use a separate class to load DDS files into our application. Direct3D implements the DDS file format for storing uncompressed or compressed textures. The file format implements several slightly different types designed for storing different types of data, and supports single layer textures, textures with mipmaps, cube maps, volume maps and texture arrays.

A sample texture and associated specular and normal maps can be found on blackboard for use in this step.

# Step 4: Loading a DDS Texture

1. Download and add the DDSTextureLoader class to your solution (this class can be found on blackboard). This class is created by Microsoft and combines the loading of DDS files along with the creation of a shader resource view. Therefore adding textures to your application is a simple process. More information on the DDSTextureLoader can be found [here](https://directxtex.codeplex.com/wikipage?title=DDSTextureLoader).
2. Add a #include for the texture loader into your main application.
3. Add a new member level variable in your application to hold the loaded texture’s shader resource view:

ID3D11ShaderResourceView \* \_pTextureRV = nullptr;

1. Now the texture can be loaded using the CreateDDSTextureFromFile method as so:

CreateDDSTextureFromFile(\_pd3dDevice, L"texture.dds", nullptr, &\_pTextureRV);

1. Finally, we tell DirectX which texture we would like to use in our shader, assigning it to texture register one:

\_pImmediateContext->PSSetShaderResources(0, 1, &\_pTextureRV);

# Step 5: Defining a sampler

To read data from a texture in our HLSL code we need to define a sampler, in this tutorial we will define the sampler in our C++ code and then pass this across to our shader for use.

1. Add a new member level variable to hold the sampler:

ID3D11SamplerState \* \_pSamplerLinear = nullptr;

1. Now inside our initialisation method we can define the specifications for our sampler:

// Create the sample state

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;

sampDesc.MinLOD = 0;

sampDesc.MaxLOD = D3D11\_FLOAT32\_MAX;

pd3dDevice->CreateSamplerState(&sampDesc, &\_pSamplerLinear);

More information on the different options available for our sampler state can be found [here](http://msdn.microsoft.com/en-gb/library/windows/desktop/ff476207(v=vs.85).aspx).

1. We tell DirectX which sampler we would like to use in our shader, assigning it to sampler register one:

\_pImmediateContext->PSSetSamplers(0, 1, &\_pSamplerLinear);

You should now have a textured model in your scene.

**Additional Tasks**

**Task 1:** The use of specular maps was discussed in the lecture on texturing. Essentially we are sampling a texture to get per-pixel specular materials to use in our lighting calculations. Use the texture provided on blackboard (or create your own) to enable per-pixel specular materials. This should improve the realism of the lighting in your scene.