Tutorial 7: Texture Mapping and Constant Buffers

Collapse All

Summary

In the previous tutorial, we introduced lighting to our project. Now we will build on that by adding textures to our cube. Also, we will introduce the concept of constant buffers, and explain how you can use buffers to speed up processing by minimizing bandwidth usage.

The purpose of this tutorial is to modify the center cube to have a texture mapped onto it.

This tutorial concludes the introduction of the basic concepts in Direct3D 10. The tutorials that follow will build upon these concepts by introducing DXUT, mesh loading, as well as an example of each shader.

Source

(SDK root)\Samples\C++\Direct3D10\Tutorials\Tutorial07

Texture mapping

Texture mapping refers to the projection of a 2D image onto 3D geometry. We can think of it as wrapping a present, by placing decorative paper over an otherwise bland box. To do this, we have to specify how the points on the surface of the geometry correspond with the 2D image.

The trick is to properly align the coordinates of the model with the texture. For complex models, it is difficult to determine the coordinates for the textures by hand. Thus, 3D modeling packages generally will export models with corresponding texture coordinates. Since our example is a cube, it is easy to determine the coordinates needed to match the texture. Texture coordinates are defined at the vertices, and are then interpolated for individual pixels on a surface.

Creating a Shader Resource from the Texture

The texture is a 2D image that is retrieved from file and used to create a shader-resource view, so that it can be read from a shader.

```
hr = D3DX10CreateShaderResourceViewFromFile( g_pd3dDevice, L"seafloor.dds", NULL, NULL,
         &g_pTextureRV, NULL );
```

Defining the Coordinates

Before we can map the image onto our cube, we must first define the texture coordinates on each of the vertices of the cube. Since images can be of any size, the coordinate system used has been normalized to [0, 1]. The top left corner of the texture corresponds to (0,0) and the bottom right corner maps to (1,1).

In this example, we're having the whole texture spread across each side of the cube. This simplifies the definition of the coordinates, without confusion. However, it is entirely possible to specify the texture to stretch across all six faces, although it's more difficult to define the points, and it will appear stretched and distorted.

First, we updated the structure used to define our vertices to include the texture coordinates.

```
struct SimpleVertex
{
    D3DXVECTOR3 Pos; // Position
    D3DXVECTOR2 Tex; // Texture Coordinate
};
```

Next, we updated the input layout to the shaders to also include these coordinates.

```
// Define the input layout
D3D10_INPUT_ELEMENT_DESC layout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
        D3D10_INPUT_PER_VERTEX_DATA, 0 },
};
```

Since the input layout changed, the corresponding vertex shader input must also be modified to match the addition.

```
struct VS_INPUT
{
    float4 Pos : POSITION;
    float2 Tex : TEXCOORD;
};
```

Finally, we are ready to include texture coordinates in our vertices we defined back in tutorial 4. Note the second parameter input is a D3DXVECTOR2 containing the texture coordinates. Each vertex on the cube will correspond to a corner of the texture. This creates a simple mapping where each vertex gets (0,0) (0,1) (1,0) or (1,1) as the coordinate.

```
// Create vertex buffer
SimpleVertex vertices[] =
{
    { D3DXVECTOR3( -1.0f,  1.0f, -1.0f ), D3DXVECTOR2( 0.0f, 0.0f ) },
    { D3DXVECTOR3(  1.0f,  1.0f, -1.0f ), D3DXVECTOR2( 1.0f, 0.0f ) },
    { D3DXVECTOR3(  1.0f,  1.0f,  1.0f ), D3DXVECTOR2( 1.0f, 1.0f ) },
    { D3DXVECTOR3( -1.0f,  1.0f,  1.0f ), D3DXVECTOR2( 0.0f, 1.0f ) },

    { D3DXVECTOR3( -1.0f, -1.0f, -1.0f ), D3DXVECTOR2( 0.0f, 0.0f ) },
    { D3DXVECTOR3(  1.0f, -1.0f, -1.0f ), D3DXVECTOR2( 1.0f, 0.0f ) },
    { D3DXVECTOR3(  1.0f, -1.0f,  1.0f ), D3DXVECTOR2( 1.0f, 1.0f ) },
    { D3DXVECTOR3( -1.0f, -1.0f,  1.0f ), D3DXVECTOR2( 0.0f, 1.0f ) },

    { D3DXVECTOR3( -1.0f, -1.0f,  1.0f ), D3DXVECTOR2( 0.0f, 0.0f ) },
    { D3DXVECTOR3( -1.0f, -1.0f, -1.0f ), D3DXVECTOR2( 1.0f, 0.0f ) },
    { D3DXVECTOR3( -1.0f,  1.0f, -1.0f ), D3DXVECTOR2( 1.0f, 1.0f ) },
    { D3DXVECTOR3( -1.0f,  1.0f,  1.0f ), D3DXVECTOR2( 0.0f, 1.0f ) },

    { D3DXVECTOR3( 1.0f, -1.0f,  1.0f ), D3DXVECTOR2( 0.0f, 0.0f ) },
    { D3DXVECTOR3( 1.0f, -1.0f, -1.0f ), D3DXVECTOR2( 1.0f, 0.0f ) },
    { D3DXVECTOR3( 1.0f,  1.0f, -1.0f ), D3DXVECTOR2( 1.0f, 1.0f ) },
    { D3DXVECTOR3( 1.0f,  1.0f,  1.0f ), D3DXVECTOR2( 0.0f, 1.0f ) },

    { D3DXVECTOR3( -1.0f, -1.0f, -1.0f ), D3DXVECTOR2( 0.0f, 0.0f ) },
    { D3DXVECTOR3(  1.0f, -1.0f, -1.0f ), D3DXVECTOR2( 1.0f, 0.0f ) },
    { D3DXVECTOR3(  1.0f,  1.0f, -1.0f ), D3DXVECTOR2( 1.0f, 1.0f ) },
    { D3DXVECTOR3( -1.0f,  1.0f, -1.0f ), D3DXVECTOR2( 0.0f, 1.0f ) },

    { D3DXVECTOR3( -1.0f, -1.0f, 1.0f ), D3DXVECTOR2( 0.0f, 0.0f ) },
    { D3DXVECTOR3(  1.0f, -1.0f, 1.0f ), D3DXVECTOR2( 1.0f, 0.0f ) },
    { D3DXVECTOR3(  1.0f,  1.0f, 1.0f ), D3DXVECTOR2( 1.0f, 1.0f ) },
    { D3DXVECTOR3( -1.0f,  1.0f, 1.0f ), D3DXVECTOR2( 0.0f, 1.0f ) },
};
```

When we sample the texture, we will need to modulate it with a material color for the geometry underneath.

Bind Texture as Shader Resource

A texture is an object like the matrices and vectors that we have seen in previous tutorials. Before they can be used by the shader, they need to be set to the Effect. You can do this by getting a pointer to the variable, and then setting it as a shader resource.

```
g_pDiffuseVariable =
    g_pEffect->GetVariableByName("txDiffuse")->AsShaderResource();
```

After you get the resource pointer, use it to hook the 2D texture resource view that we had initialized earlier.

```
g_pDiffuseVariable->SetResource( g_pTextureRV );
```

There we go, now we're ready to use the texture within the shader.

Applying the Texture (fx)

To actually map the texture on top of the geometry, we will be calling a texture lookup function within the pixel shader. The function, Sample, will perform a texture lookup of a 2D texture, and then return the sampled color. The pixel shader shown below calls this function and multiplies it by the underlying mesh color (or material color), and then outputs the final color.

- txDiffuse is the object storing our texture that we passed in from the code above, when we bound the resource view g_pTextureRV to it.
- samLinear will be described below; it is the sampler specifications for the texture lookup.
- input.Tex is the coordinates of the texture that we have specified in the source.

```
// Pixel Shader
float4 PS( PS_INPUT input) : SV_Target
{
    return txDiffuse.Sample( samLinear, input.Tex ) * vMeshColor;
}
```

The variable *samLinear* is a structure that contains the information to tell the pixel shader how to sample the texture provided. In our case, we have a linear filter, and both our addresses wrap. These settings would be useful for simple textures. An explanation of filters is beyond the scope of this tutorial.

```
SamplerState samLinear
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};
```

Another thing we must remember to do is to pass the texture coordinates through the vertex shader. If we don't, the data is lost when it gets to the pixel shader. Here, we just copy the input's coordinates to the output, and let the hardware handle the rest.

```
// Vertex Shader
PS_INPUT VS( VS_INPUT input )
{
    PS_INPUT output = (PS_INPUT)0;
    output.Pos = mul( input.Pos, World );
    output.Pos = mul( output.Pos, View );
    output.Pos = mul( output.Pos, Projection );
    output.Tex = input.Tex;

    return output;
}
```

Constant Buffers

Starting with Direct3D 10, an application can use a **constant buffer** to set shader constants (shader variables). Constant buffers are declared using a syntax similar to C-style structs. Constant buffers reduce the bandwidth required to update shader constants by allowing shader constants to be grouped together and committed at the same time, rather than making individual calls to commit each constant separately.

The best way to efficiently use constant buffers is to organize shader variables into constant buffers based on their frequency of update. This allows an application to minimize the bandwidth required for updating shader constants. As an example, this tutorial groups constants into three structures: one for variables that change every frame, one for variables that change only when a window size is changed, and one for variables that are set once and then do not change.

The following constant buffers are defined in this tutorial's .fx file.

```
    cbuffer cbNeverChanges
    {
        matrix View;
    };

    cbuffer cbChangeOnResize
    {
        matrix Projection;
```

```
};

cbuffer cbChangesEveryFrame
{
    matrix World;
    float4 vMeshColor;
};
```

This tutorial uses an ID3D10Effect instance to manage the update of the constant buffers when their variables change. The ID3D10Effect interface makes many tasks easier, including the creation, update, and setting of constant buffers. The following code shows you the use of constant buffer variables in the application.

Getting the constant buffer variables from the effect instance causes the effect instance to set up any required constant buffers.

```
//called once after the effect is loaded and the ID3D10Effect instance is created
g_pWorldVariable = g_pEffect->GetVariableByName( "World" )->AsMatrix();
g_pMeshColorVariable = g_pEffect->GetVariableByName( "vMeshColor" )->AsVector();
```

Updating the constant buffer variables signals the effect instance to commit the constant buffers at the appropriate times.

```
//called whenever the variables in the constant buffer are changed
g_pWorldVariable->SetMatrix( ( float* )&g_World );
g_pMeshColorVariable->SetFloatVector( ( float* )g_vMeshColor );
```