Tutorial 5: Using Texture Maps

While lights and materials add a great deal of realism to a scene, nothing adds more realism than adding textures to surfaces. Textures can be thought of as wallpaper that is shrink-wrapped onto a surface. You could place a wood texture on a cube to make the cube look like it is actually made of wood. The Texture sample project adds a banana peel texture to the cylinder created in tutorial 4. This tutorial covers how to load textures, set up vertices, and display objects with texture.

This tutorial implements textures using the following steps:

Steps

- Step 1 - Defining a Custom Vertex Format
- Step 2 - Initializing Screen Geometry
- Step 3 - Rendering the Scene

Note

The path of the Texture sample project is:

(*SDK root*)\Samples\C++\Direct3D\Tutorials\Tut05_Textures

The sample code in the Texture project is nearly identical to the sample code in the Lights project, except that the Texture sample project does not create a material or a light. The Using Texture Maps tutorial focuses only on the code unique to textures and does not cover initializing Direct3D, handling Windows messages, rendering, or shutting down. For information about these tasks, see Tutorial 1: Creating a Device.

This tutorial uses custom vertices and a vertex buffer to display geometry. For more information about selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.

This tutorial makes use of matrices to transform geometry. For more information about matrices and transformations, see Tutorial 3: Using Matrices.

Step 1 - Defining a Custom Vertex Format

Before using textures, a custom vertex format that includes texture coordinates must be used. Texture coordinates tell Direct3D where to place a texture for each vector in a primitive. Texture coordinates range from 0.0 to 1.0, where (0.0, 0.0) represents the top-left side of the texture and (1.0, 1.0) represents the lower-right side of the texture.

The following sample code shows how the Texture sample project sets up its custom vertex format to include texture coordinates.

```
// A structure for our custom vertex type. Texture coordinates were added.
struct CUSTOMVERTEX
{
    D3DXVECTOR3 position; // The position
    D3DCOLOR    color;    // The color
#ifndef SHOW_HOW_TO_USE_TCI
    FLOAT       tu, tv;   // The texture coordinates
#endif
};

// Custom flexible vertex format (FVF), which describes custom vertex structure
#ifdef SHOW_HOW_TO_USE_TCI
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE)
```

```
#else
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX1)
#endif
```

For more information about texture coordinates, see **Texture Coordinate Formats (Direct3D 9)**.

Now that a custom vertex type has been defined, the next step is to load a texture and create a cylinder, as described in Step 2 - Initializing Screen Geometry.

Step 2 - Initializing Screen Geometry

Before rendering, the Texture sample project calls InitGeometry, an application-defined function that creates a texture and initializes the geometry for a cylinder.

Textures are created from file-based images. The following code fragment uses **D3DXCreateTextureFromFile** to create a texture from Banana.bmp that will be used to cover the surface of the cylinder.

```
if( FAILED( D3DXCreateTextureFromFile( g_pd3dDevice, "Banana.bmp",
&g_pTexture ) ) )
return E_FAIL;
```

The first parameter that **D3DXCreateTextureFromFile** accepts is a pointer to the Direct3D device that will be used to render the texture. The second parameter is a pointer to an ANSI string that specifies the filename from which to create the texture. This sample specifies Banana.bmp to load the image from that file. The third parameter is the address of a pointer to a texture object.

When the banana texture is loaded and ready to use, the next step is to create the cylinder. The following code sample fills the vertex buffer with a cylinder. Note that each point has the texture coordinates (tu, tv).

```
for( DWORD i=0; i<50; i++ )
{
    FLOAT theta = (2*D3DX_PI*i)/(50-1);

        pVertices[2*i+0].position = D3DXVECTOR3( sinf(theta),-1.0f, cosf(theta) );
        pVertices[2*i+0].color    = 0xffffffff;
#ifndef SHOW_HOW_TO_USE_TCI
        pVertices[2*i+0].tu       = ((FLOAT)i)/(50-1);
        pVertices[2*i+0].tv       = 1.0f;
#endif

        pVertices[2*i+1].position = D3DXVECTOR3( sinf(theta), 1.0f, cosf(theta) );
        pVertices[2*i+1].color    = 0xff808080;
#ifndef SHOW_HOW_TO_USE_TCI
        pVertices[2*i+1].tu       = ((FLOAT)i)/(50-1);
        pVertices[2*i+1].tv       = 0.0f;
#endif
}
```

Each vertex includes position, color, and texture coordinates. The code sample above sets the texture coordinates for each point so that the texture will wrap smoothly around the cylinder.

Now that the texture is loaded and the vertex buffer is ready for rendering, it is time to render the display, as described in Step 3 - Rendering the Scene.

Step 3 - Rendering the Scene

After scene geometry has been initialized, it is time to render the scene. In order to render an object with texture, the texture must be set as one of the current textures. The next step is to set the texture stage states values. Texture stage states enable you to define the behavior of how a texture or textures are to be rendered. For example, you could blend multiple textures together.

The Texture sample project starts by setting the texture to use. The following code fragment sets the texture that the Direct3D device will use to render with **IDirect3DDevice9::SetTexture**.

```
g_pd3dDevice->SetTexture( 0, g_pTexture );
```

The first parameter that **IDirect3DDevice9::SetTexture** accepts is a stage identifier to set the texture to. A device can have up to eight set textures, so the maximum value here is 7. The Texture sample project has only one texture and places it at stage 0. The second parameter is a pointer to a texture object. The Texture sample project uses the texture object that it created in its InitGeometry application-defined function.

The following code sample sets the texture stage state values by calling the **IDirect3DDevice9::SetTextureStageState** method.

```
    // Setup texture. Using textures introduces the texture stage states, which
    //   govern how textures get blended together (in the case of multiple
    //   textures) and lighting information. In this case, you are modulating
    //   (blending) your texture with the diffuse color of the vertices.
    g_pd3dDevice->SetTexture( 0, g_pTexture );
    g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_MODULATE );
    g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
    g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
    g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP,   D3DTOP_DISABLE );

#ifdef SHOW_HOW_TO_USE_TCI
    // Note: To use Direct3D texture coordinate generation, use the stage state
    // D3DTSS_TEXCOORDINDEX, as shown below. In this example, you are using
    // the position of the vertex in camera space to generate texture
    // coordinates. The tex coord index (TCI) parameters are passed into a
    // texture transform, which is a 4x4 matrix that transforms the x,y,z
    // TCI coordinates into tu, tv texture coordinates.

    // In this example, the texture matrix is set up to
    // transform the texture from (-1,+1) position coordinates to (0,1)
    // texture coordinate space:
    //    tu =  0.5*x + 0.5
    //    tv = -0.5*y + 0.5
    D3DXMATRIXA16 mat;
    mat._11 = 0.25f; mat._12 = 0.00f; mat._13 = 0.00f; mat._14 = 0.00f;
    mat._21 = 0.00f; mat._22 =-0.25f; mat._23 = 0.00f; mat._24 = 0.00f;
    mat._31 = 0.00f; mat._32 = 0.00f; mat._33 = 1.00f; mat._34 = 0.00f;
    mat._41 = 0.50f; mat._42 = 0.50f; mat._43 = 0.00f; mat._44 = 1.00f;

    g_pd3dDevice->SetTransform( D3DTS_TEXTURE0, &mat );
    g_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
                  D3DTTFF_COUNT2 );
    g_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX,
                  D3DTSS_TCI_CAMERASPACEPOSITION );
#endif
```

The first parameter that **IDirect3DDevice9::SetTextureStageState** accepts is the stage index for the state variable to be set. This code sample changes the values for the texture at stage 0, so it puts a zero here. The next parameter is the texture state to set. For a list of all valid texture states and their meaning, see **D3DTEXTURESTAGESTATETYPE**. The next parameter is the value to which the texture state will be set. The value that you place here is based on the texture stage state value that you are modifying.

After setting up the desired values for each texture stage state, the cylinder can be rendered and texture will be added to the surface.

Another way to use texture coordinates is to automatically generate them. This is done by using a texture coordinate index (TCI). The TCI uses a texture matrix to transform the (x,y,z) TCI coordinates into (tu, tv) texture coordinates. In the Texture sample project, the position of the vertex in camera space is used to

generate texture coordinates.

The first step is to create the matrix that will be used for the transformation, as demonstrated in the following code fragment.

```
D3DXMATRIX mat;
mat._11 = 0.25f; mat._12 = 0.00f; mat._13 = 0.00f; mat._14 = 0.00f;
mat._21 = 0.00f; mat._22 =-0.25f; mat._23 = 0.00f; mat._24 = 0.00f;
mat._31 = 0.00f; mat._32 = 0.00f; mat._33 = 1.00f; mat._34 = 0.00f;
mat._41 = 0.50f; mat._42 = 0.50f; mat._43 = 0.00f; mat._44 = 1.00f;
```

After the matrix is created, it must be set by calling **IDirect3DDevice9::SetTransform**, as shown in the following code fragment.

```
g_pd3dDevice->SetTransform( D3DTS_TEXTURE0, &mat );
```

The D3DTS_TEXTURE0 flag tells Direct3D to apply the transformation to the texture located at texture stage 0. The next step that this sample takes is to set more texture stage state values to get the desired effect. That is done in the following code fragment.

```
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
                            D3DTTFF_COUNT2 );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX,
                            D3DTSS_TCI_CAMERASPACEPOSITION );
```

The texture coordinates are set up, and now the scene is ready to be rendered. Notice that the coordinates are automatically created for the cylinder. This particular setup gives the effect of the texture being laid over the rendering screen after the geometric shapes have been rendered.

For more information about textures, see **Direct3D Textures (Direct3D 9)**.