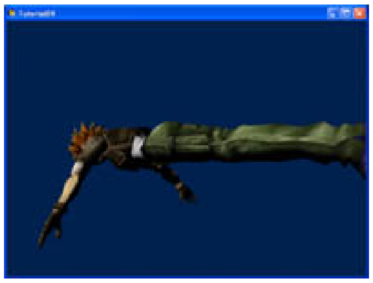
Tutorial 9: Meshes in DXUT

Collapse All



Summary

This tutorial introduces the use of meshes to import source art. It demonstrates meshes by using DXUT. However, meshes can also be used without DXUT.

In the tutorial, you replace a cube in the center of the window with a model that is imported from a file. The model already contains a texture with mapped coordinates.

Source

(SDK root)\Samples\C++\Direct3D10\Tutorials\Tutorial09

Meshes

Listing each vertex within the source code is a tedious and crude method to define source art. Instead, there are methods to import already constructed models into your application.

A mesh is a collection of predefined vertex data which often includes additional information such as normal vectors, colors, materials, and texture coordinates. As long as the format of the mesh file is known, the mesh file can be imported and rendered.

Meshes are used to maintain source art separately from application code, so that the art can be reused. Therefore, meshes are stored in separate files. They are rarely generated in an application itself. Instead, they are created in 3D authoring software. The cube that was generated in previous tutorials can be saved into a separate file, and then re-loaded. However, importing can be much more efficient, especially if the mesh scales in complexity. One of the biggest benefits is the ability to import the texture coordinates, so that they match up perfectly. These can easily be specified in authoring applications.

DXUT handles meshes by using the CDXUTMesh10 class, which is a wrapper for the D3DX Mesh class. This class includes functions to import, render, and destroy a mesh. The file format we use for importing is the .X file. There are many freely available converters that convert other model formats to the .X format.

Creating the Mesh

To import the model, we first create a CDXUTMesh10 object. It is called g_Mesh in this case, but in general the object name and the source art name should match, for easier association.

```
CDXUTMesh10 g_Mesh;
```

Next, we call the Create function to read the X file and store it into the object. The Create function requires that we pass in the D3D10Device, the name of the X file holding our mesh, the layout, and the number of elements in the layout. There is also an optional parameter called optimize, which calls D3DX functions to reorder the faces and vertices of a given mesh. This can improve rendering performance.

In this case, we import a file that is called "tiny.x". The format for the vertices that define this mesh contains vertex coordinates, normals, and texture coordinates. We must specify our input layout to match, as shown in the following example.

```
// Define the input layout
const D3D10_INPUT_ELEMENT_DESC layout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"TEXCOORD0", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24, D3D10_INPUT_PER_VERTEX_DATA, 0 },
};
```

After the layout is defined, we call the Create function to import the model. The last parameter is the number of elements in the vertex format. Because we have coordinates, normal, and texture coordinates, we specify 3.

```
// Load the mesh
```

```
    V_RETURN( g_Mesh.Create( pd3dDevice, L"tiny.x", (D3D10_INPUT_ELEMENT_DESC*)layout, 3 ) );
```

If there are no errors, g_Mesh now contains the vertex and index buffer for our newly imported mesh, as well as textures and materials. Next we begin rendering.

Rendering the Mesh

In the previous tutorials, because we had explicit control of the vertex buffer and the index buffers, we had to set them up properly before every frame. However, with a mesh, these elements are abstracted. Therefore, we only have to provide the effects technique that should be used to render the mesh, and it does all the work.

The difference from the previous tutorial is that we removed the portion of OnD3DFrameRender where it began rendering the cube. We replaced it with the mesh rendering call. It is always good practice to set the correct input layout before any mesh is rendered. This ensures that the layout of the mesh matches the input assembler.

```
    //
    // Set the Vertex Layout
    //
    pd3dDevice->IASetInputLayout( g_pVertexLayout );
```

The actual rendering is done by calling the Render function inside the CDXUTMesh10 class. After the technique's buffers are correctly associated, it can be rendered. To render, we pass in the D3D10Device, the Effect, and the Technique within that effect.

```
    //
    // Render the mesh
    //
    g_Mesh.Render( pd3dDevice, g_pEffect, g_pTechnique );
```

If the world matrix is properly set for the model, we can now see that the cube in our previous example is replaced by this more complicated mesh. Notice that the mesh is much bigger than the cube. Try applying a scale to the world matrix to get the mesh to fit the screen better.

Destroying the Mesh

Like all objects, the CDXUTMesh10 object must be destroyed after usage. This is done by calling the Destroy function.

```
    g_Mesh.Destroy();
```