Tutorial 6: Using Meshes

Complicated geometry is usually modeled using 3D modeling software, after which the model is saved to a file. An example of this is the .x file format. Direct3D uses meshes to load the objects from these files. Meshes are somewhat complicated, but D3DX contains functions that make using meshes easier. The Meshes sample project introduces the topic of meshes and shows how to load, render, and unload a mesh.

This tutorial shows how to load, render, and unload a mesh using the following steps.

Steps

- Step 1 - Loading a Mesh Object
- Step 2 - Rendering a Mesh Object
- Step 3 - Unloading a Mesh Object

The path of the Meshes sample project is: (SDK root)\Samples\C++\Direct3D\Tutorials\Tut06_Meshes

The sample code in the Meshes project is nearly identical to the sample code in the Lights project, except that the code in the Meshes project does not create a material or a light. This tutorial focuses only on the code unique to meshes and does not cover setting up Direct3D, handling Windows messages, rendering, or shutting down.

Step 1 - Loading a Mesh Object

A Direct3D application must first load a mesh before using it. The Meshes sample project loads the tiger mesh by calling InitGeometry>, an application-defined function, after loading the required Direct3D objects.

A mesh needs a material buffer that will store all the materials and textures that will be used. The function starts by declaring a material buffer as shown in the following code fragment.

```
LPD3DXBUFFER pD3DXMtrlBuffer;
```

The following code fragment loads the mesh.

```
// Load the mesh from the specified file
if( FAILED( D3DXLoadMeshFromX( "Tiger.x", D3DXMESH_SYSTEMMEM,
            g_pd3dDevice, NULL, &pD3DXMtrlBuffer, NULL,
            &g_dwNumMaterials, &g_pMesh ) ) )
    {
    // If model is not in current folder, try parent folder
    if( FAILED( D3DXLoadMeshFromX( "..\\Tiger.x",
            D3DXMESH_SYSTEMMEM, g_pd3dDevice, NULL,
            &pD3DXMtrlBuffer, NULL, &g_dwNumMaterials,
            &g_pMesh ) ) )
        {
            MessageBox(NULL, "Could not find tiger.x",
                       "Meshes.exe", MB_OK);
            return E_FAIL;
        }
    }
```

The first parameter is a pointer to a string that tells the name of the DirectX file to load. This sample loads the tiger mesh from Tiger.x.

The second parameter specifies how to create the mesh. The sample uses the D3DXMESH_SYSTEMMEM flag, which is equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM. Both of these flags put the index buffer and vertex buffer for the mesh in system memory.

The third parameter is a pointer to a device that will be used to render the mesh.

The fourth parameter is a pointer to an **ID3DXBuffer** object. This object will be filled with information about neighbors for each face. This information is not required for this sample, so this parameter is set to NULL.

The fifth parameter also takes a pointer to an **ID3DXBuffer** object. After this method is finished, this object will be filled with D3DXMATERIAL structures for the mesh.

The sixth parameter is a pointer to the number of D3DXMATERIAL structures placed into the *ppMaterials* array after the method returns.

The seventh parameter is the address of a pointer to a mesh object, representing the loaded mesh.

After loading the mesh object and material information, you need to extract the material properties and texture names from the material buffer.

The Meshes sample project does this by first getting the pointer to the material buffer. The following code fragment uses the **ID3DXBuffer::GetBufferPointer** method to get this pointer.

```
D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
```

The following code fragment creates new mesh and texture objects based on the total number of materials for the mesh.

```
g_pMeshMaterials = new D3DMATERIAL9[g_dwNumMaterials];
g_pMeshTextures  = new LPDIRECT3DTEXTURE9[g_dwNumMaterials];
```

For each material in the mesh the following steps occur.

The first step is to copy the material, as shown in the following code fragment.

```
g_pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
```

The second step is to set the ambient color for the material, as shown in the following code fragment.

```
g_pMeshMaterials[i].Ambient = g_pMeshMaterials[i].Diffuse;
```

The final step is to create the texture for the material, as shown in the following code fragment.

```
// Create the texture.
if( FAILED( D3DXCreateTextureFromFile( g_pd3dDevice,
   d3dxMaterials[i].pTextureFilename, &g_pMeshTextures[i] ) ) )
    g_pMeshTextures[i] = NULL;
}
```

After loading each material, you are finished with the material buffer and need to release it by calling IUnknown.

```
pD3DXMtrlBuffer->Release();
```

The mesh, along with the corresponding materials and textures are loaded. The mesh is ready to be rendered to the display, as described in Step 2 - Rendering a Mesh Object.

Step 2 - Rendering a Mesh Object

In step 1 the mesh was loaded and is now ready to be rendered. It is divided into a subset for each material that was loaded for the mesh. To render each subset, the mesh is rendered in a loop. The first step in the loop is to set the material for the subset, as shown in the following code fragment.

```
g_pd3dDevice->SetMaterial( &g_pMeshMaterials[i] );
```

The second step in the loop is to set the texture for the subset, as shown in the following code fragment.

```
g_pd3dDevice->SetTexture( 0, g_pMeshTextures[i] );
```

After setting the material and texture, the subset is drawn with the **ID3DXBaseMesh::DrawSubset** method, as shown in the following code fragment.

```
g_pMesh->DrawSubset( i );
```

The **ID3DXBaseMesh::DrawSubset** method takes a DWORD that specifies which subset of the mesh to draw. This sample uses a value that is incremented each time the loop runs.

After using a mesh, it is important to properly remove the mesh from memory, as described in Step 3 - Unloading a Mesh Object.

Step 3 - Unloading a Mesh Object

After any DirectX program finishes, it needs to deallocate any DirectX objects that it used and invalidate the pointers to them. The mesh objects used in this sample also need to be deallocated. When it receives a WM_DESTROY message, the Meshes sample project calls Cleanup, an application-defined function, to handle this.

The following code fragment deletes the material list.

```
if(g_pMeshMaterials)
delete[] g_pMeshMaterials;
```

The following code fragment deallocates each individual texture that was loaded and then deletes the texture list.

```
if(g_pMeshTextures)
{
    for(DWORD i = 0; i < g_dwNumMaterials; i++
    {
        if(g_pMeshTextures[i])
            g_pMeshTextures[i]->Release();
    }
delete[] g_pMeshTextures;
```

The following code fragment deallocates the mesh object.

```
// Delete the mesh object
if(g_pMesh)
    g_pMesh->Release();
```

This tutorial has shown you how to load and render meshes. This is the last tutorial in this section.