MeshFromObj Sample

Collapse All

This sample shows how an **ID3DXMesh** object can be created from mesh data stored in a Wavefront Object file (.obj). It's convenient to use .x files when working with **ID3DXMesh** objects, since D3DX can create and fill a mesh object directly from a .x file. It's also easy to initialize a mesh object with data gathered from any file format or memory resource.

Path

Source	*SDK root*\Samples\C++\Direct3D\MeshFromObj
Executable	*SDK root*\Samples\C++\Direct3D\Bin*x86 or x64*\MeshFromObj.exe

Sample Overview

The **ID3DXMesh** interface is a handy way to manage complex meshes. It allows you to divide the mesh into subsets, usually according to separate material types. When rendering the mesh you can loop through subsets, setting the proper render state for the current material. The interface treats the complex mesh as a single object while still offering the flexibility of rendering with several different textures or shaders.

The interface implements member functions for optimizing how internal mesh data is sorted. This leads to fewer state changes during rendering and increased vertex cache performance.

ID3DXMesh can encapsulate any mesh that can be rendered by using indexed triangle lists, and therefore, it does not rely on any particular file format. The purpose of this sample is to show how an **ID3DXMesh** object can be created from arbitrary mesh data.

Implementation

ID3DXMesh objects contain a vertex buffer and an associated index buffer. Each **ID3DXMesh** object is rendered as an indexed triangle list, so a call to **DrawSubset** is functionally the same as retrieving the internal vertex and index buffers and calling **IDirect3DDevice9::DrawIndexedPrimitive** using the D3DPT_TRIANGLELIST primitive type. To fill an **ID3DXMesh** object with mesh data you simply need to fill the internal vertex and index buffers. When creating a mesh directly from a .x file using **D3DXLoadMeshFromX**, setting the vertex format and populating the vertex and index buffers are handled automatically. If you use a different mesh file format (than .x) you will need to set up the vertex and index buffers yourself.

For this sample, the .obj format was chosen for its simplicity, but the lesson extends to any file format or memory resource layout. In fact, the most important snippet of sample code is the portion which deals with loading an **ID3DXMesh** object once the mesh data has already been imported:

```
// Create the encapsulated mesh
ID3DXMesh* pMesh = NULL;
V_RETURN( D3DXCreateMesh( m_Indices.GetSize() / 3, m_Vertices.GetSize(),
D3DXMESH_MANAGED | D3DXMESH_32BIT, VERTEX_DECL,
pd3dDevice, &pMesh ) );

// Copy the vertex data
VERTEX* pVertex;
V_RETURN( pMesh->LockVertexBuffer( 0, (void**) &pVertex ) );
memcpy( pVertex, m_Vertices.GetData(), m_Vertices.GetSize() * sizeof(VERTEX) );
pMesh->UnlockVertexBuffer();
m_Vertices.RemoveAll();

// Copy the index data
DWORD* pIndex;
V_RETURN( pMesh->LockIndexBuffer( 0, (void**) &pIndex ) );
memcpy( pIndex, m_Indices.GetData(), m_Indices.GetSize() * sizeof(DWORD) );
pMesh->UnlockIndexBuffer();
m_Indices.RemoveAll();
```

```
// Copy the attribute data
DWORD* pSubset;
V_RETURN( pMesh->LockAttributeBuffer( 0, &pSubset ) );
memcpy( pSubset, m_Attributes.GetData(), m_Attributes.GetSize() * sizeof(DWORD) );
pMesh->UnlockAttributeBuffer();
m_Attributes.RemoveAll();
```

The code above locks and fills the internal vertex and index buffers, much the same way these buffers would be accessed if used independently of the **ID3DXMesh** interface. This code also fills the optional attribute buffer. Each face in the mesh can be given an attribute, which is the number of the subset to which it belongs; during a call to **ID3DXMesh::DrawSubset(DWORD** *AttribId* **)**, those faces with an attribute that matches the attribute ID will be rendered. If rendering the entire mesh, the application simply needs to loop through all the subsets. The attribute IDs divide a mesh into groups which are rendered separately. For this sample, the mesh is divided into material groups which allows an application to render each subset with the correct material properties.

The following code renders a subset of the mesh; it shows how the texture, illumination variables, and shaders are adjusted for the particular material prior to rendering the subset:

```
void RenderSubset( UINT iSubset )
{
HRESULT hr;
UINT iPass, cPasses;

// Retrieve the ID3DXMesh pointer and current material from the MeshLoader helper
ID3DXMesh* pMesh = g_MeshLoader.GetMesh();
Material* pMaterial = g_MeshLoader.GetMaterial( iSubset );

// Set the lighting variables and texture for the current material
V( g_pEffect->SetValue( g_hAmbient, pMaterial->vAmbient, sizeof(D3DXVECTOR3) ) );
V( g_pEffect->SetValue( g_hDiffuse, pMaterial->vDiffuse, sizeof(D3DXVECTOR3) ) );
V( g_pEffect->SetValue( g_hSpecular, pMaterial->vSpecular, sizeof(D3DXVECTOR3) ) );
V( g_pEffect->SetTexture( g_hTexture, pMaterial->pTexture ) );
V( g_pEffect->SetFloat( g_hOpacity, pMaterial->fAlpha ) );
V( g_pEffect->SetFloat( g_hSpecularPower, pMaterial->fShininess ) );

V( g_pEffect->SetTechnique( pMaterial->hTechnique ) );
V( g_pEffect->Begin(&cPasses, 0) );

for (iPass = 0; iPass < cPasses; iPass++)
{
V( g_pEffect->BeginPass(iPass) );

// Render the mesh with the applied technique
V( pMesh->DrawSubset( iSubset ) );

V( g_pEffect->EndPass() );
}
V( g_pEffect->End() );
}
```

The next section demonstrates how to import mesh data from a .obj file. Note that the simple parser created for this sample is not intended to be a complete parser for the .obj format nor any of its extensions.

Loading the Geometry from a File

Wavefront Object (.obj) files are an ASCII format for describing meshes composed of triangle lists. Vertices may contain texture coordinates and normals. Object files may also contain a reference to a Material Library (.mtl) file, which defines materials containing illumination variables and texture file names. Mesh faces can be assigned a material to allow separate components of the mesh to retain individual material properties. Each line of the file begins with a token which indicates the type of information stored on that line; the possible tokens are shown in the following table.

Token	Comment (ignored)
v float float float	Vertex position v (position x) (position y) (position z)
vn float float	Vertex normal vn (normal x) (normal y) (normal z)

float	
vt float float	Texture coordinate vt (tu) (tv)
f int int int	f (v) (v) (v) Faces are stored as a series of three vertices in clockwise order. Vertices are described by their position, optional texture coordinate, and optional normal, encoded as indices into the respective component lists.
f int/int int/int int/int	f (v)/(vt) (v)/(vt) (v)/(vt)
f int/int/int int/int/int int/int/int	f (v)/(vt)/(vn) (v)/(vt)/(vn) (v)/(vt)/(vn)
mtllib string	Material file (MTL) mtllib (*file-name*.mtl) References the MTL file for this mesh. MTL files contain illumination variables and texture file names.
usemtl string	Material selection usemtl (material name) Faces which are listed after this point in the file will use the selected material

The mesh files are parsed within the **CMeshLoader::LoadGeometryFromOBJ**, which builds a vertex buffer, an index buffer, and an attribute buffer. After the file is completely parsed, these buffers can be copied into an empty **ID3DXMesh** object. **LoadGeometryFromOBJ** builds vertex position, texture coordinate, and vertex normal lists directly from file input. When a mesh face element is encountered within the input file, the vertex components are retrieved from the respective component lists.

Since multiple vertices may share the same position but have different texture coordinates or vertex normals, a little extra work must be done when storing vertices, otherwise duplicate vertices will be added to the vertex buffer (in which case there's no point to having an index buffer). To make this check relatively quick, vertices are stored in a hash table cache according to their position index as they are added to the vertex buffer. This cache is checked for duplicates by **CMeshLoader::AddVertex** for each new face. After the mesh has been completely loaded from file, this vertex cache is destroyed by calling the **CMeshLoader::DeleteCache**.

Most .obj files also reference a .mtl file, which contains named material sets. The parser builds a list of materials it expects to find within the .mtl file, and keeps track of which material is selected as each mesh face is read. This material index is also used as the attribute number for a given face; these attribute numbers divide the mesh faces into material type subsets. Although the material groupings may be scattered throughout the .obj file, the vertex buffer will be reordered into material subsets as part of the call to **ID3DXMesh::Optimize**, so no additional sorting is required of the parser.

Loading the Materials from a File

If an MTL file was referenced within the object file, it's parsed immediately after the object file. The MTL file format uses an ASCII encoding very similar to .obj files, using the tokens shown in the following table.

Token	Comment (ignored)
newmtl string	Material newmtl (material name). Begins a new material description.
Ka float float float	Ambient color Ka (red) (green) (blue)
Kd float float float	Diffuse color Kd (red) (green) (blue)
Ks float float float	Specular color Ks (red) (green) (blue)
d float Tr float	Transparency Tr (alpha)
Ns int	Shininess Ns (specular power)
illum int	Illumination model illum (1 / 2); 1 if specular disabled, 2 if specular enabled.
map_Kd string	Texture map_Kd (file name)

When the .obj file is parsed and new material names are encountered, material structures containing default values are created and added to the material list. When the .mtl file is parsed, these structures are retrieved from the material list and filled with the correct values. In addition, a default material is created to contain all the faces which aren't otherwise assigned a material. These conventions guarantee that all faces will be assigned a material and missing material descriptions will be filled with default values.

Texture file names are stored within the material structures, and the corresponding textures are created upon device creation. When creating textures, the material list is scanned for duplicate texture filenames to ensure that materials which use the same texture point to a single created instance. After the .mtl file has been parsed, each item in the material list will correspond to a subset in the **ID3DXMesh** object. As described above, this material list is iterated through in combination with corresponding mesh subsets when rendering the entire mesh.

© 2010 Microsoft Corporation. All rights reserved.
Send feedback to DxSdkDoc@microsoft.com.
Version: 1962.00