# Chapter 19

# D3DX Mesh Objects

"**net**, *n.*: Anything reticulated or decussated at equal distances, with interstices between the intersections"
Samuel Johnson: *Dictionary*, 1755

## 19.1 Overview

A surface modelled as a collection of adjacent triangles is referred to as a triangle mesh. When drawn in wireframe mode, such a triangulated surface has an appearance similar to a fishing net. D3DX provides several COM objects and their associated global functions that encapsulate meshes as indexed triangle lists. The interface hierarchy of the mesh objects in D3DX is given in figure figure 19.1.

A base mesh interface, `ID3DXBaseMesh`, handles the details of managing the vertex and index buffers needed for the mesh. A derived interface, `ID3DXMesh`, is the most commonly used mesh interface and provides optimized rendering of the triangles in the mesh.

D3DX provides objects that allow you to implement a level of detail scheme through mesh simplification and progressive meshes. Mesh simplification is the process of reducing the number of vertices or triangles in a mesh, while preserving its overall shape and appearance. View independent mesh simplification reduces the number of vertices or triangles based on global attributes of the mesh. View dependent simplification techniques take into account the orientation and position of the mesh within the scene to simplify small distant meshes more aggressively, while large meshes in the foreground are more detailed. D3DX provides the `ID3DXSPMesh` interface and several functions for view independent mesh simplification.

Progressive meshes take the idea of mesh simplification further and provide a rapid means of obtaining a mesh at any vertex or triangle size within a range of sizes. With progressive meshes, dynamic level of detail on a mesh can rapidly

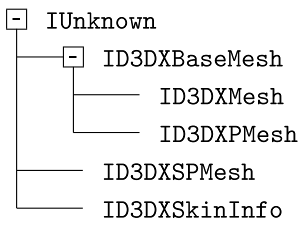


Figure 19.1: D3DX Mesh Interface Hierarchy.

be achieved, while mesh simplification is more suitable to static level of detail refinements due to the cost of computing the simplified mesh. D3DX provides the `ID3DXPMesh` object for progressive meshes.

A skinned mesh combines a triangle mesh with vertex blending to give the appearance of a smoothly deforming object. D3DX provides the `ID3DXSkinInfo` object to create meshes with skinning information.

## 19.2 Use of `const` Qualifiers

Unfortunately, the D3DX mesh header file improperly uses `const` qualifiers in conjunction with pointer `typedef`s in several places. The intention was to declare pointers to arrays of constant data, but the declarations are written so as to produce a type signature of a constant pointer to an array of data. The problem comes from the use of a `typedef` alias for a pointer to the type in combination with `const`.

To understand the problem, consider the following function declaration. The code declares the `data` argument to be a pointer to an array of `const DWORD`s. The intention is to signify that the function will not modify the array of data, even though its being passed to the function as a pointer.

```
void foo(const DWORD *data, DWORD size);
```

Now suppose we change the declaration to use a `typedef` for the `DWORD` pointer. The Windows standard header `<windows.h>` defines a `typedef` for a pointer to `DWORD` called `LPDWORD` through the included file `<windef.h>`.

```
void foo(const LPDWORD data, DWORD size);
```

The problem here is that `LPDWORD` isn't a macro, but an alias for another type. If it were a macro, then the substitution would occur as you might expect and `data` would still declare a pointer to a `const` array. However, because `LP-DWORD` is a type alias for a pointer to `DWORD`, the code now declares the pointer to be `const` rather than declaring the target of the pointer to be `const`.

There are two ways to correct the problem. The first way to handle this is to avoid the `typedef` for pointers completely as in the first declaration above.

The second way is to introduce additional typedefs for the const variants. The Windows header files do this for some, but not all, of the type aliases defined.

This unfortunate misuse of const affects the following methods and functions.

| **Methods** | |
|---|---|
| ID3DXSkinMesh::ConvertToBlendedMesh | |
| ID3DXSkinMesh::ConvertToIndexedBlendedMesh | |
| ID3DXSkinMesh::GenerateSkinnedMesh | |
| **Functions** | |
| ::D3DXCreateSPMesh | ::D3DXSimplifyMesh |
| ::D3DXGeneratePMesh | ::D3DXSplitMesh |
| ::D3DXSaveMeshToX | ::D3DXWeldVertices |

When you use these methods or functions, you may encounter the following compiler error:

```
error C2664: 'D3DXSaveMeshToX' : cannot convert parameter 4 from
'const struct D3DXMATERIAL *' to 'struct D3DXMATERIAL *const '
```

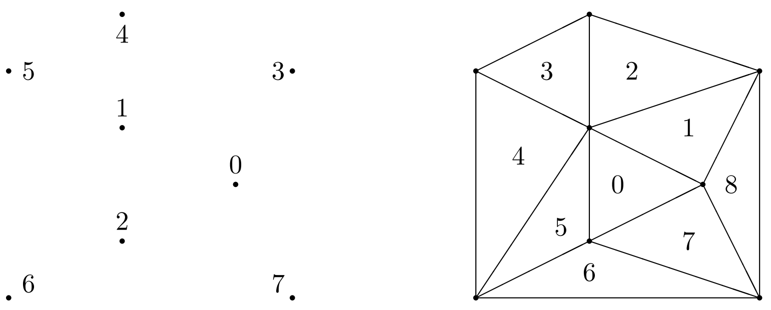
You can eliminate the error with the const_cast<> casting operator on the offending parameter. In this particular case, the code can be written as follows.

```
THR(::D3DXSaveMeshToX("foo2.x", mesh, &adj[0],
    const_cast<D3DXMATERIAL *const>(&mats[0]), NUM_OF(mats),
    DXFILEFORMAT_TEXT));
```

Fortunately, this problem only appears in the <d3dx9mesh.h> header file and only for a handful of methods and functions. However, it can be irritating if you use these methods or functions often. If that is the case, you may wish to create your own inline wrapper around the function to eliminate the need for const_cast<>. For methods you can declare an inline function that takes the interface pointer and performs the method call with the appropriate cast.

For instance, the following wrapper around D3DXSaveMeshToX eliminates the need for the const_cast<> for the materials parameter and also allows the filename parameter to be passed in as const. (The function does not modify the filename, but does not declare the string as const.)

```
namespace rt
{
    HRESULT SaveMeshToX(LPCSTR filename, ID3DXMesh *mesh,
        const DWORD *adjacency, const D3DXMATERIAL *materials,
        DWORD num_materials, DWORD format)
    {
        return ::D3DXSaveMeshToX(const_cast<LPSTR>(filename),
            mesh, adjacency,
            const_cast<D3DXMATERIAL *const>(materials),
```



| Vertex | Position |
|---|---|
| 0 | $(0, 0, 0)$ |
| 1 | $(-2, 1, 0)$ |
| 2 | $(-2, -1, 0)$ |
| 3 | $(1, 2, 0)$ |
| 4 | $(-2, 3, 0)$ |
| 5 | $(-4, 2, 0)$ |
| 6 | $(-4, -2, 0)$ |
| 7 | $(1, -2, 0)$ |

| Triangle | Indices | Adjacency |
|---|---|---|
| 0 | $1, 0, 2$ | $1, 7, 5$ |
| 1 | $1, 3, 0$ | $2, 8, 0$ |
| 2 | $4, 3, 1$ | $-1, 1, 3$ |
| 3 | $5, 4, 1$ | $-1, 2, 4$ |
| 4 | $5, 1, 6$ | $3, 5, -1$ |
| 5 | $1, 2, 6$ | $0, 6, 4$ |
| 6 | $2, 7, 6$ | $7, -1, 5$ |
| 7 | $0, 7, 2$ | $8, 6, 0$ |
| 8 | $3, 7, 0$ | $-1, 7, 1$ |

Figure 19.2: Triangle Adjacency. The vertices on the left are arranged into the triangles on the right, forming a mesh of triangles in the $xy$ plane.

```
            num_materials, format);
    }
};
```

The header file `<rt/const.h>` in the sample code includes wrappers around methods and functions mentioned in this section as well as a few other places where pointers to `const` strings should have been used, making D3DX a bit easier to use with the standard library string classes.

## 19.3 Triangle Adjacencies

Many mesh algorithms make use of triangle adjacency information. The adjacency of a triangle gives the triangles adjacent to each of its edges. With an indexed triangle list, each triangle is stored as three vertex indices in the index buffer. Each triangle is numbered with an unsigned integer, starting from zero. The offset into the index buffer for a triangle is three times its triangle number. The adjacency information corresponds to the order in which the vertices are listed for each triangle.

For example, consider the simple planar triangle mesh shown in figure 19.2. For triangle number zero, the vertex indices are $1, 0, 2$ with adjacency $1, 7, 5$. As can be seen from the figure, triangle one is adjacent to the edge made by vertices 1, 0, triangle seven is adjacent to the edge made by vertices 0, 2 and triangle five is adjacent to the edge made by vertices 2, 1.

When there is no adjacent triangle, the adjacency value for that edge is the `DWORD` value `0xFFFFFFFF`. When interpreted as a 32-bit integer, this is the value −1, used in figure 19.2.

When D3DX methods or functions pass adjacency information, they expect to receive or return arrays of `DWORD`s, three for each triangle. The number of triangles expected to be in the array is the same as the corresponding number of triangles in the particular mesh.

## 19.4 Vertex and Triangle Remapping

Mesh object methods and functions may reorder the vertices and triangles within a mesh. When this occurs, the functions return arrays of `DWORD`s indicating the new position of vertices or triangles within the vertex and index buffers. For example, if the mesh in figure 19.2 were optimized to reorder the vertices and triangles in triangle strip order, then the resulting face and vertex remap arrays would be as follows.

| Vertex | Remapped To |
|---|---|
| 0 | 3 |
| 1 | 7 |
| 2 | 0 |
| 3 | 2 |
| 4 | 6 |
| 5 | 1 |
| 6 | 4 |
| 7 | 5 |

| Triangle | Remapped To |
|---|---|
| 0 | 8 |
| 1 | 7 |
| 2 | 6 |
| 3 | 5 |
| 4 | 0 |
| 5 | 1 |
| 6 | 2 |
| 7 | 3 |
| 8 | 4 |

The values in the left-hand column of each table give the index into the remap array and the values in the right-hand column give the value stored at that array location. So, if the remap arrays are declared as standard vector classes as:

```
std::vector<DWORD> vertmap(mesh->GetNumVertices());
std::vector<DWORD> trimap(mesh->GetNumFaces());
```

then the new location in the vertex buffer for vertex 3 is given by `vertmap[3]` and has the value 2. Similarly, the new location in the index buffer for the indices of triangle 3 is given by `trimap[3]` and has the value 5. Since there are three indices for each triangle, the triangle position corresponds to the indices at offsets 15, 16, and 17.

When vertices or triangles are removed from a mesh, the corresponding remap value will be the `DWORD` value `0xFFFFFFFF`. When interpreted as a 32-bit integer, this is the value −1.

## 19.5 Base Mesh

The `ID3DXBaseMesh` interface is the base interface for `ID3DXMesh` and `ID3DXP-Mesh`. It provides operations common to both mesh objects for accessing information relating to the number of triangles in the mesh and their related vertex and index data. The interface is summarized in interface 19.1. A mesh stores the geometry data as indexed triangle list primitives. The triangles can be grouped into subsets, with all triangles in the subset sharing the same rendering state. This allows the mesh objects to be ignorant of the necessary device state, while still allowing for it.

Interface 19.1: Summary of the `ID3DXBaseMesh` interface.

| **ID3DXBaseMesh** | |
|---|---|
| **Read-Only Properties** | |
| `GetAttributeTable` | The attribute table entries |
| `GetDeclaration` | The vertex shader declaration for the mesh vertices |
| `GetDevice` | The associated device |
| `GetFVF` | The FVF code for the mesh vertices |
| `GetIndexBuffer` | The indices of the mesh |
| `GetNumFaces` | The number of triangles in the mesh |
| `GetNumVertices` | The number of vertices in the mesh |
| `GetOptions` | The mesh options |
| `GetVertexBuffer` | The vertices of the mesh |
| **Methods** | |
| `CloneMesh` | Makes a copy of the mesh with a declarator |
| `CloneMeshFVF` | Makes a copy of the mesh with an FVF code |
| `ConvertAdjacencyTo-PointReps` | Create adjacencies from point representatives |
| `ConvertPointRepsTo-Adjacency` | Create point representatives from adjacencies |
| `DrawSubset` | Renders triangles with common attributes |
| `GenerateAdjacency` | Generates adjacency information |
| `LockIndexBuffer` | Obtain access to the indices of the mesh |
| `LockVertexBuffer` | Obtain access to the vertices of the mesh |
| `UnlockIndexBuffer` | Release access to the indices of the mesh |
| `UnlockVertexBuffer` | Release access to the vertices of the mesh |

```
interface ID3DXBaseMesh : IUnknown
{
    // read-only properties
    HRESULT GetAttributeTable(D3DXATTRIBUTERANGE *value,
                DWORD *size);
    HRESULT GetDeclaration(DWORD value[MAX_FVF_DECL_SIZE]);
    HRESULT GetDevice(IDirect3DDevice9 **value);
    DWORD GetFVF();
    HRESULT GetIndexBuffer(IDirect3DIndexBuffer9 **value);
    DWORD GetNumFaces();
    DWORD GetNumVertices();
    DWORD GetOptions();
    HRESULT GetVertexBuffer(IDirect3DVertexBuffer9 **value);

    // methods
    HRESULT CloneMeshFVF(DWORD options,
                DWORD fvf,
                IDirect3DDevice9 *device,
                ID3DXMesh **result);
    HRESULT CloneMesh(DWORD options,
                const DWORD *declaration,
                IDirect3DDevice9 *device,
                ID3DXMesh **result);
    HRESULT ConvertAdjacencyToPointReps(const DWORD *adjacency,
                DWORD *pPRep);
    HRESULT ConvertPointRepsToAdjacency(const DWORD *pPRep,
                DWORD *adjacency);
    HRESULT DrawSubset(DWORD attribute);
    HRESULT GenerateAdjacency(float Epsilon,
                DWORD *adjacency);
    HRESULT LockIndexBuffer(DWORD Flags,
                BYTE **ppData);
    HRESULT LockVertexBuffer(DWORD Flags,
                BYTE **ppData);
    HRESULT UnlockIndexBuffer();
    HRESULT UnlockVertexBuffer();
};
```

The device property returns the device associated with the mesh's index and vertex buffers and is returned by `GetDevice`. The vertex and index buffers themselves are returned by the `GetVertexBuffer` and `GetIndexBuffer` methods, respectively. The number of triangles and vertices in the mesh are returned by the `GetNumFaces` and `GetNumVertices` methods.

TODO: xbaseGetDeclaration, xbaseGetFVF

The format of the vertices can be retrieved either as an FVF code by the `GetFVF` method or as a vertex declaration by the `GetDeclaration` method. The mesh objects provided by D3DX only support meshes that can be described with an FVF code. The declaration related methods and functions are only provided as a convenience. When obtaining the format of the vertex as a declaration, an array of `DWORD`s is passed into `GetDeclaration`. The size of the array is declared to be `MAX_FVF_DECL_SIZE`.

```
enum _MAX_FVF_DECL_SIZE
```

```
{
    MAX_FVF_DECL_SIZE = 20
};
```

The options property of the mesh is obtained by the `GetOptions` method and describes the usage pattern of the associated vertex and index buffers containing the mesh data. The options property is a `DWORD` containing zero or more of the `D3DXMESH` flags, shown here in four groups. The first group contains flags relating to the mesh as a whole. The second and third groups contain flags relating to the usage of the mesh's index and vertex buffers. The fourth group combines the corresponding flags from the second and third groups, giving the usage of both buffers. A mesh in the default pool is created by omitting the managed and system memory options.

```
enum _D3DXMESH
{
    // general flags
    D3DXMESH_32BIT                 = 0x00001,
    D3DXMESH_DONOTCLIP             = 0x00002,
    D3DXMESH_NPATCHES              = 0x04000,
    D3DXMESH_POINTS                = 0x00004,
    D3DXMESH_RTPATCHES             = 0x00008,
    D3DXMESH_USEHWONLY             = 0x02000,

    // index buffer usage flags
    D3DXMESH_IB_DYNAMIC            = 0x00800,
    D3DXMESH_IB_MANAGED            = 0x00200,
    D3DXMESH_IB_SOFTWAREPROCESSING = 0x10000,
    D3DXMESH_IB_SYSTEMMEM          = 0x00100,
    D3DXMESH_IB_WRITEONLY          = 0x00400,

    // vertex buffer usage flags
    D3DXMESH_VB_DYNAMIC            = 0x00080,
    D3DXMESH_VB_MANAGED            = 0x00020,
    D3DXMESH_VB_SOFTWAREPROCESSING = 0x08000,
    D3DXMESH_VB_SYSTEMMEM          = 0x00010,
    D3DXMESH_VB_WRITEONLY          = 0x00040,
    D3DXMESH_VB_SHARE              = 0x01000,

    // combined usage flags
    D3DXMESH_MANAGED               = 0x00220,
    D3DXMESH_DYNAMIC               = 0x00880,
    D3DXMESH_SOFTWAREPROCESSING    = 0x18000,
    D3DXMESH_SYSTEMMEM             = 0x00110,
    D3DXMESH_WRITEONLY             = 0x00440
};
```

The attribute table divides the triangles in the mesh into subsets. The table is created when one of the optimize methods of ID3DXMesh is used to sort the mesh data by the attribute ids in the attribute buffer. If the attribute buffer is modified, then the attribute table is cleared.

Subsets are usually used to group triangles together that share common rendering state. However, as far as the mesh objects are concerned, subsets are merely groups of triangles; the mesh objects themselves are not concerned with the rendering state. For example, the application could use subsets to divide the mesh into portions that are hidden, higlighted, selected, and so-on.

The GetAttributeTable method returns the attribute table as an array of D3DXATTRIBUTERANGE structures. The value argument may be NULL, in which case the number of entries in the entire table will be stored in the size argument. Otherwise, the size argument gives the number of structures that can be stored in the value argument.

```
typedef struct _D3DXATTRIBUTERANGE
{
    DWORD AttribId;
    DWORD FaceStart;
    DWORD FaceCount;
    DWORD VertexStart;
    DWORD VertexCount;
} D3DXATTRIBUTERANGE;
```

The AttribId member is an arbitrary value that identifies the subset described by the structure. The range of indices associated with this subset are given by the FaceStart and FaceCount members. Triangles are numbered from zero in the order they are stored in the index buffer, with groups of three indices identifying the three vertices used for each triangle. The VertexStart and VertexCount members identify the range of vertices associated with the subset. Unless the triangles in the mesh are sorted by their attribute id, the triangles and vertices associated with the subset may not be contiguous in the associated buffers.

In common Windows style, the method will typically be called twice: once to find out the size of the table so that appropriate storage can be allocated and a second time to obtain the table itself. For example, the following code obtains the attribute table from an ID3DXBaseMesh interface declared as base.

```
DWORD size = 0;
THR(base->GetAttributeTable(NULL, &size));
std::vector<D3DXATTRIBUTERANGE> table(size);
THR(base->GetAttributeTable(&table[0], &size));
```

A copy, or clone, of the mesh can be used to add or remove vertex components to the mesh, or change the options associated with the mesh. The vertex components of the clone are described by either an FVF code, with the CloneMeshFVF method, or by a vertex declaration with the CloneMesh method. The vertex declaration must correspond to a valid FVF code, or the method will fail. The `options` parameter to these methods gives the `D3DXMESH` flags for the cloned mesh. For instance, the following code clones a system memory mesh in the variable `mesh` into a mesh in the default pool.

```
CComPtr<ID3DXMesh> clone;
THR(mesh->CloneMeshFVF(
    mesh->GetOptions() & ~(D3DXMESH_MANAGED | D3DXMESH_SYSTEMMEM),
    mesh->GetFVF(), device, &clone));
```

The index and vertex buffers associated with the mesh can be accessed directly with the `LockIndexBuffer`, `UnlockIndexBuffer`, `LockVertexBuffer`, and `UnlockVertexBuffer` methods. These methods function similarly to the corresponding lock and unlock methods on the underlying vertex and index buffers. Helper functions for ensuring that unlock is always called for each successful lock can be constructed for mesh objects, as was done in chapter 5. Such helper classes are defined in the include file `<rt/mesh.h>` in the sample code.

The triangle adjacency information can be generated from the index and vertex buffer contents directly by the `GenerateAdjacency` method. This is useful when another mesh operation requires adjacency information and the vertex and index buffers were filled directly by the application. The `epsilon` parameter gives a tolerance for comparing vertex positions to determine if they represent the same position. The `adjacency` parameter points to an array of `DWORD`s and should contain at least three `DWORD`s for each triangle in the mesh. For example, the following generates the adjacency for a mesh into a local array, considering vertices whose position coordinates differ by less than $10^{-6}$ to be identical.

```
std::vector<DWORD> adjacency(mesh->GetNumFaces()*3);
THR(mesh->GenerateAdjacency(1.0e-6f, &adjacency[0]));
```

A point representative is another way of thinking about the adjacency information of a mesh. With a point representative, each vertex in the mesh is associated with another vertex that is the representative point for the vertex. Some model formats use a single vertex position associated with multiple sets of other vertex components, such as texture coordinates, normals, etc. These formats use independent indices for the different vertex components, while Direct3D uses a unique index for each vertex as a whole. When converting from such a format to Direct3D, the original vertex position associated with the components becomes duplicated for each differing set of vertex components. The duplicated positions refer to the original position through the point representative index. The `ConvertAdjacencyToPointReps` and `ConvertPointRepsToAdjacency` methods provide a means for converting from a point representative array and an adjacency array.

For example, consider a flat shaded cube with surface normals, such as the mesh created with `D3DXCreateBox`. Such a cube will have 24 vertices, due to differing surface normals at each corner for the six faces of the cube. The

| Vertex | Point Representative | Position | Normal |
|---|---|---|---|
| 0 | 0 | $(-1,-1,-1)$ | $\langle -1, 0, 0\rangle$ |
| 1 | 1 | $(-1,-1,1)$ | $\langle -1, 0, 0\rangle$ |
| 2 | 2 | $(-1,1,1)$ | $\langle -1, 0, 0\rangle$ |
| 3 | 3 | $(-1,1,-1)$ | $\langle -1, 0, 0\rangle$ |
| 4 | 3 | $(-1,1,-1)$ | $\langle 0, 1, 0\rangle$ |
| 5 | 2 | $(-1,1,1)$ | $\langle 0, 1, 0\rangle$ |
| 6 | 6 | $(1,1,1)$ | $\langle 0, 1, 0\rangle$ |
| 7 | 7 | $(1,1,-1)$ | $\langle 0, 1, 0\rangle$ |
| 8 | 7 | $(1,1,-1)$ | $\langle 1, 0, 0\rangle$ |
| 9 | 6 | $(1,1,1)$ | $\langle 1, 0, 0\rangle$ |
| 10 | 10 | $(1,-1,1)$ | $\langle 1, 0, 0\rangle$ |
| 11 | 11 | $(1,-1,-1)$ | $\langle 1, 0, 0\rangle$ |
| 12 | 1 | $(-1,-1,1)$ | $\langle 0, -1, 0\rangle$ |
| 13 | 0 | $(-1,-1,-1)$ | $\langle 0, -1, 0\rangle$ |
| 14 | 11 | $(1,-1,-1)$ | $\langle 0, -1, 0\rangle$ |
| 15 | 10 | $(1,-1,1)$ | $\langle 0, -1, 0\rangle$ |
| 16 | 1 | $(-1,-1,1)$ | $\langle 0, 0, 1\rangle$ |
| 17 | 10 | $(1,-1,1)$ | $\langle 0, 0, 1\rangle$ |
| 18 | 6 | $(1,1,1)$ | $\langle 0, 0, 1\rangle$ |
| 19 | 2 | $(-1,1,1)$ | $\langle 0, 0, 1\rangle$ |
| 20 | 0 | $(-1,-1,-1)$ | $\langle 0, 0, -1\rangle$ |
| 21 | 3 | $(-1,1,-1)$ | $\langle 0, 0, -1\rangle$ |
| 22 | 7 | $(1,1,-1)$ | $\langle 0, 0, -1\rangle$ |
| 23 | 11 | $(1,-1,-1)$ | $\langle 0, 0, -1\rangle$ |

Table 19.1: Vertex data for a flat shaded cube.

| Face | Vertex Indices | Edge Adjacencies |
|---|---|---|
| 0 | 0, 1, 2 | 6, 9, 1 |
| 1 | 2, 3, 0 | 2, 10, 0 |
| 2 | 4, 5, 6 | 1, 9, 3 |
| 3 | 6, 7, 4 | 4, 10, 2 |
| 4 | 8, 9, 10 | 3, 8, 5 |
| 5 | 10, 11, 8 | 7, 11, 4 |
| 6 | 12, 13, 14 | 0, 11, 7 |
| 7 | 14, 15, 12 | 5, 8, 6 |
| 8 | 16, 17, 18 | 7, 4, 9 |
| 9 | 18, 19, 16 | 2, 0, 8 |
| 10 | 20, 21, 22 | 1, 3, 11 |
| 11 | 22, 23, 20 | 5, 6, 10 |

Table 19.2: Face data for a flat shaded cube.

cube is a completely closed surface, so each face will have adjacent faces on all three edges. The associated vertex and face data are shown in table 19.1 and table 19.2.

The final method in the base mesh interface is `DrawSubset`. This method sets the minimum amount of device state necessary to drawn an indexed triangle list: the mesh's vertex and index buffers are set as the source for primitive data and a vertex shader is set. Then `DrawPrimitive` is called for the appropriate triangles in the subset. The `attribute` argument identifies the subset to render. If no attribute table exists for the mesh, then all the triangles are categorized into subset zero.

`DrawSubset` uses a fixed-function vertex shader based on the FVF of the mesh's vertices. If you wish to draw a mesh subset with a programmable vertex shader, you can use the information in the attribute buffer to draw the appropriate triangles. The following code snippet shows how to call `DrawIndexedPrimitive` for the triangles in a mesh. The calls to `set_state` and `set_default_state` are only illustrative of the location where any subset-specific rendering state should be set based on its attribute identifier.

```
void
render_mesh(DWORD shader, ID3DXBaseMesh *mesh)
{
    CComPtr<IDirect3DDevice9> device;
    THR(mesh->GetDevice(&device));

    CComPtr<IDirect3DVertexBuffer9> vb;
    THR(mesh->GetVertexBuffer(&vb));
    THR(device->SetStreamSource(0, vb,
        ::D3DXGetFVFVertexSize(mesh->GetFVF())));

    CComPtr<IDirect3DIndexBuffer9> ib;
    THR(mesh->GetIndexBuffer(&ib));
    THR(device->SetIndices(ib, 0));

    THR(device->SetVertexShader(shader));

    DWORD size = 0;
    THR(mesh->GetAttributeTable(NULL, &size));
    if (size)
    {
        std::vector<D3DXATTRIBUTERANGE> table(size);
        THR(mesh->GetAttributeTable(&table[0], &size));

        for (UINT i = 0; i < size; i++)
        {
            set_state(device, table[i].Attribid);
            THR(device->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
                table[i].VertexStart, table[i].VertexCount,
                table[i].FaceStart*3, table[i].FaceCount));
        }
    }
    else
    {
        set_default_state();
        THR(device->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
            0, mesh->GetNumVertices(), 0, mesh->GetNumFaces()));
    }
}
```

## 19.6 Mesh

The most commonly used mesh interface is ID3DXMesh, which provides direct access to the attribute buffer and mesh optimization. A mesh object can be created with D3DXCreateMeshFVF or D3DXCreateMesh from a vertex FVF code or vertex declaration, respectively. The options parameter specifies D3DXMESH options for the encapsulated vertex and index buffers. The ID3DXMesh interface is summarized in interface 19.2.

```
HRESULT  D3DXCreateMeshFVF(DWORD num_faces,
            DWORD num_vertices,
            DWORD options,
            DWORD fvf,
            IDirect3DDevice9 *device,
            ID3DXMesh **result);

HRESULT D3DXCreateMesh(DWORD num_faces,
            DWORD num_vertices,
            DWORD options,
            const DWORD *declaration,
            IDirect3DDevice9 *device,
            ID3DXMesh **result);
```

Interface 19.2: Summary of the ID3DXMesh interface.

**ID3DXMesh**

| **Methods** | |
|---|---|
| LockAttributeBuffer | Obtains direct access to the attribute table |
| Optimize | Optimizes mesh vertices into a new mesh |
| OptimizeInplace | Optimizes mesh vertices into the existing mesh |
| UnlockAttributeBuffer | Releases direct access to the attribute table |

```
interface ID3DXMesh : ID3DXBaseMesh
{
    // methods
    HRESULT LockAttributeBuffer(DWORD flags,
                DWORD **ppData);
    HRESULT Optimize(DWORD flags,
                const DWORD *adjacency_in,
                DWORD *adjacency_out,
                DWORD *face_remap,
                ID3DXBuffer **vertex_remap,
                ID3DXMesh **result);
    HRESULT OptimizeInplace(DWORD flags,
                const DWORD *adjacency_in,
                DWORD *adjacency_out,
                DWORD *face_remap,
                ID3DXBuffer **vertex_remap);
    HRESULT UnlockAttributeBuffer();
};
```

The attribute buffer can be accessed directly with the `LockAttributeBuffer` and `UnlockAttributeBuffer` methods. The `flags` argument can be one of the values `D3DLOCK_DISCARD`, `D3DLOCK_NOOVERWRITE`, `D3DLOCK_NOSYSLOCK`, or `D3D-LOCK_READONLY`. The flags have the same meaning as those used for locking a vertex buffer, described on page 178. The attribute buffer is exposed as an array of `DWORD` values, one per triangle, that indicate the attribute identifier of the triangle.

The `OptimizeInplace` and `Optimize` methods reorder vertices and indices to optimize rendering performance. The data is reordered based on the `flags` argument, which may be zero or more of the following values.

```
enum _D3DXMESHOPT
{
    D3DXMESHOPT_ATTRSORT     = 0x02000000,
    D3DXMESHOPT_COMPACT      = 0x01000000,
    D3DXMESHOPT_IGNOREVERTS  = 0x10000000,
    D3DXMESHOPT_SHAREVB      = 0x00001000,
    D3DXMESHOPT_STRIPREORDER = 0x08000000,
    D3DXMESHOPT_VERTEXCACHE  = 0x04000000
};
```

The attribute sort flag reorders indices so that triangles sharing common rendering state can be rendered with fewer calls to `DrawIndexedPrimitive`. The compact flag reorders indices to remove unused vertices and faces. The underlying vertex and index buffers are not made smaller, the unused data is moved to the end of the buffers. The ignore vertices flag indicates that only the indices should be modified and the vertices left unchanged. The share vertex buffer flag indicates that any resulting mesh should share the vertex buffer with the original mesh.

The last two flags are mutually exclusive and indicate the ordering of indices to improve the rendering performance of individual `DrawIndexedPrimitive` calls. The strip reorder flag reorders indices to render triangles in the same order as the set of largest possible triangle strips. The vertex cache flag reorders the indices to increase the hardware vertex cache hit rate. The cache optimization is specific to the hardware from different vendors and is implemented with access to information on the hardware covered by non-disclosure agreements between Microsoft and the hardware vendors.

If you wish to take advantage of the cache optimization, but don't wish to use D3DX mesh interfaces, you can still take advantage of cache reordering optimization facilities. You can create a mesh object with the appropriate number of vertices and indices and copy your mesh data into the object's buffers. Then call `OptimizeInplace` with the appropriate flags and copy the vertex and index buffer data back into your own buffers.

## 19.6.1 Creating Meshes From Scratch

Using a combination of the mesh helper classes in `<rt/mesh.h>` from the sample code and the standard library, a mesh can be constructed from scratch and filled with data in a small amount of code. The following code constructs a mesh containing a cube in the `InitDeviceObjects` method of an application generated with the sample framework. Each face of the cube is modelled as two triangles, each in its own subset. Each vertex contains a position and a surface normal to provide flat shading for the cube. The data for the vertices and indices are given in table 19.1 and table 19.2 and are omitted from the code listing for clarity.

```
#define NUM_OF(ary_) (sizeof(ary_)/sizeof((ary_)[0]))

struct s_vertex
{
    float x, y, z;
    float nx, ny, nz;

    static const DWORD FVF;
};

const DWORD s_vertex::FVF = D3DFVF_XYZ | D3DFVF_NORMAL;

HRESULT CMyD3DApplication::InitDeviceObjects()
{
    const s_vertex vertices[24] =
    {
        // ....
    };
    const WORD indices[12*3] =
    {
        // ...
    };
    const DWORD attributes[12] =
    {
        0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5
    };

    THR(::D3DXCreateMeshFVF(NUM_OF(indices)/3, NUM_OF(vertices),
        D3DXMESH_MANAGED, s_vertex::FVF, m_pd3dDevice, &m_mesh));

    std::copy(vertices, vertices + NUM_OF(vertices),
              rt::mesh_vertex_lock<s_vertex>(m_mesh).data());
    std::copy(indices, indices + NUM_OF(indices),
              rt::mesh_index_lock<WORD>(m_mesh).data());
    std::copy(attributes, attributes + NUM_OF(attributes),
              rt::mesh_attribute_lock(m_mesh).data());

    std::vector<DWORD> adjacency(m_mesh->GetNumFaces()*3);
    THR(m_mesh->GenerateAdjacency(0.0f, &adjacency[0]));

    THR(m_mesh->OptimizeInplace(D3DXMESHOPT_VERTEXCACHE |
        D3DXMESHOPT_ATTRSORT, &adjacency[0], NULL, NULL, NULL));

    return S_OK;
}
```

The mesh is created and filled with data in four statements. The standard copy algorithm is used to copy the data from arrays into the corresponding buffers using the locking helper classes. The lock is constructed as an unnamed temporary, locking the underlying buffer, and the data pointer is passed to the copy algorithm. The unnamed temporary will be destroyed when the copy algorithm returns and the lock object's destructor will call unlock on the associated buffer. This code is just as efficient as if we had written all the locking by hand, but the helper class allows us to be more concise and productive. This style of programming may seem odd or even look like a jumble of punctuation at first, but after a short while you will recognize this idiom quickly for what it does.

Three more statements optimize the mesh for rendering. The adjacency array for the mesh is generated and then used to optimize the mesh for vertex cache order and to sort the triangles by attribute. Altogether, the code for constructing and optimizing the mesh object is only a few lines.

### 19.6.2 Primitive Construction

D3DX provides routines for creating some simple geometric shapes, as well as the classic "Utah Teapot" model[1]. Each of these functions returns an `ID3DXMesh` instance and an adjacency array stored in an `ID3DXBuffer` interface. All of the meshes are described by vertices containing a position and a surface normal in a left-handed coordinate system. If additional vertex components are required, such as texture coordinates, the mesh should be cloned to include those components. The `rt::dx_buffer` helper class, described in section 15.18 on page 596 simplifies the use of the `ID3DXBuffer` interface with meshes.

`D3DXCreatePolygon` creates an $n$-sided polygon. The `length` parameter gives the length in model coordinates of each side of the polygon and the `sides` parameter gives the number of sides. The plane of the polygon is aligned with the $xy$ plane and the surface normal of the polygon faces the $+z$ direction. The polygon is centered on the origin.

```
HRESULT D3DXCreatePolygon(IDirect3DDevice9 *device,
            float length,
            UINT sides,
            ID3DXMesh **result,
            ID3DXBuffer **adjacency);
```

`D3DXCreateBox` creates a rectangular parallelepiped or box. The sides of the box are aligned with the three coordinate axes. The length of the box along the $x$, $y$, and $z$ axes are given by the `width`, `height`, and `depth` parameters, respectively. The box is centered on the origin.

```
HRESULT D3DXCreateBox(IDirect3DDevice9 *device,
            float width,
            float height,
            float depth,
            ID3DXMesh **result,
            ID3DXBuffer **adjacency);
```

`D3DXCreateCylinder` creates a polygonal approximation to a cylinder. The axis of the cylinder is aligned with and centered along the $z$ axis. The cylinder may be tapered by specifying different radii at each end of the cylinder. The radius at the $-z$ and $+z$ ends are given by the `radius1` and `radius2` parameters, respectively. The length of the cylinder along the $z$ axis is given by the `length` parameter. The cylinder is constructed from stacks along the $z$ axis. Each stack is made up of slices along the radial circumference of the stack. The number of stacks in the cylinder is given by the `stacks` parameter. The number of slices in each stack is given by the `slices` parameter.

```
HRESULT D3DXCreateCylinder(IDirect3DDevice9 *device,
            float radius1,
```

[1]See section 19.12 for an article on the origin of the teapot.

```
            float radius2,
            float length,
            UINT slices,
            UINT stacks,
            ID3DXMesh **result,
            ID3DXBuffer **adjacency);
```

D3DXCreateSphere creates a polygonal approximation to a sphere. The sphere is centered on the origin and has a radius given by the radius parameter. The sphere is approximated as a series of stacks paralell to the $xy$ plane. Each stack is made up of slices about the $z$ axis. The number of stacks in the sphere is given by the stacks parameter and the number of of slices within each stack is given by the slices parameter.

```
HRESULT D3DXCreateSphere(IDirect3DDevice9 *device,
            float radius,
            UINT slices,
            UINT stacks,
            ID3DXMesh **result,
            ID3DXBuffer **adjacency);
```

D3DXCreateTorus creates a polygonal approximation to a torus. The torus is centered on the origin and the loop of the torus lies in the $xy$ plane. The torus is approximated as a series of rings around the outer radius. Each ring is a strip of triangles around the inner radius. The number of rings is given by the rings parameter and the number of sides in each ring is given by the sides parameter. The outer_radius parameter gives the distance from the $z$ axis for the loop of the torus and the inner_radius parameter gives the radius of the loop itself.

```
HRESULT D3DXCreateTorus(IDirect3DDevice9 *device,
            float inner_radius,
            float outer_radius,
            UINT sides,
            UINT rings,
            ID3DXMesh **result,
            ID3DXBuffer **adjacency);
```

D3DXCreateTeapot creates a mesh containing the "Utah Teapot." The teapot is constructed from a fixed triangle list and therefore takes no parameters to define its shape.

```
HRESULT D3DXCreateTeapot(IDirect3DDevice9 *device,
            ID3DXMesh **result,
            ID3DXBuffer **adjacency);
```

D3DXCreateText creates extruded 3D solid text. The resulting mesh is centered on the origin, lies parallel to the $xy$ plane and has a thickness along the $z$

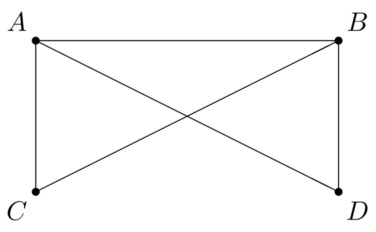


Figure 19.3: Bowtie arrangement of vertices in a triangle mesh

axis given by the `extrusion` parameter. The mesh for the text is constructed by obtaining the glyph outline for each character in the text string. The `deviation` parameter gives the maximum chordal deviation from the glyph outline and is used to determine the location of vertices approximating the smooth outline. The font used for the mesh is the font currently selected in the `dc` parameter. The `metrics` parameter returns the GDI font glyph metric data for each character in the string argument `text`.

```
HRESULT D3DXCreateText(IDirect3DDevice9 *device,
            HDC dc,
            LPCTSTR text,
            float deviation,
            float extrusion,
            ID3DXMesh **result,
            ID3DXBuffer **adjacency,
            GLYPHMETRICSFLOAT *metrics);
```

### 19.6.3 Reducing Mesh Data

D3DX provides several functions for reducing the amount of data in a mesh. Vertex components can be removed with the `CloneMesh` and `CloneMeshFVF` methods, but they don't reduce the number of triangles in the mesh. Reducing the number of triangles in a mesh is the process of mesh simplification. Meshes, particularly those obtained from modelers or other external processes, can contain certain arrangments of vertices that are problematic for simplification algorithms. The `D3DXValidMesh` function examines a mesh and its adjacency data for vertex arrangements that can cause problems with simplification algorithms. If the mesh is considered valid, then the function returns a successful `HRESULT`. If the mesh is considered invalid, then a failed `HRESULT` will be returned and the `errors` buffer will contain an error message as an ANSI string.

```
HRESULT D3DXValidMesh(ID3DXMesh *mesh,
            const DWORD *adjacency,
            ID3DXBuffer **errors);
```

Meshes will be considered invalid if a "bowtie" vertex topology is found. A bowtie exists in a triangle mesh when two separate triangle fans use the same vertex. For example, consider the vertices in figure 19.3. Triangles $\triangle ABC$ and $\triangle ABD$ share the common edge $AB$, but as part of two distinct triangle fans.

For situations where `D3DXValidMesh` fails, `D3DXCleanMesh` can be used to correct the problem in the original mesh. This consists of adding copies of existing vertices and adjusting the indices accordingly to eliminate the bowtie topology. Note that this does not change the number of triangles in the mesh, although it does change the number of vertices. The `result` parameter contains the cleansed mesh. If the adjacency of the cleansed mesh is desired, the `adjacency_out` parameter should point to an array of three `DWORD`s per triangle. If the adjacency information is not needed, this parameter may be `NULL`. If this function fails, then the mesh errors could not be corrected by `D3DXCleanMesh` and error messages describing the failure are returned as an ANSI string in the `errors` parameter. This parameter may be `NULL` if the error string is not required.

```
HRESULT D3DXCleanMesh(ID3DXMesh *mesh,
            const DWORD *adjacency_in,
            ID3DXMesh **result,
            DWORD *adjacency_out,
            ID3DXBuffer **errors);
```

The `D3DXWeldVertices` function combines vertices that are considered to be equivalent; see section 19.2 for more about the use of `const` in the `mesh` parameter to this function. Vertices are compared using $\epsilon$ values for each of the vertex components, given by the `D3DXWELDEPSILONS` structure. As duplicate vertices are removed from the mesh, the vertices of the triangles in the mesh and their adjacency may be changed. The `adjacency_in` parameter supplies the adjacency information of the original mesh and the `adjacency_out` paremeter returns the adjacency of the resulting mesh. The `face_remap` parameter points to an array of `DWORD`s, one per triangle in the original mesh, that identifies which triangles in the input mesh were mapped to triangles in the output mesh. The triangles are numbered starting from zero. The `vertex_remap` parameter returns an `ID3DXBuffer` interface containing an array of `DWORD`s, one per vertex in the original mesh, that identifies how the vertices were reordered. If the resulting adjacency or remap information is not needed, the parameters may be `NULL`.

```
HRESULT D3DXWeldVertices(ID3DXMesh *const mesh,
            D3DXWELDEPSILONS *epsilons,
            const DWORD *adjacency_in,
            DWORD *adjacency_out,
            DWORD *face_remap,
            ID3DXBuffer **vertex_remap);

typedef struct _D3DXWELDEPSILONS
{
    float SkinWeights;
```

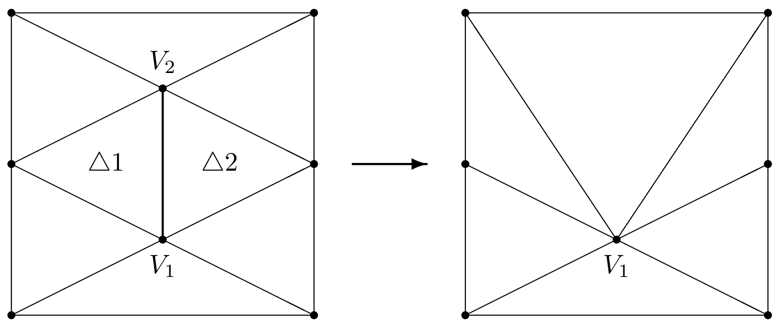


Figure 19.4: Mesh simplification by collapsing edges. The common edge $V_1V_2$ shared by triangles △1 and △2 is collapsed, eliminating the triangles sharing the edge. The vertex $V_1$ remains, allowing the edge collapse transformation to be implemented only by an adjustment to the indices of the mesh.

```
    float Normal;
    float Tex[8];
    DWORD Flags;
} D3DXWELDEPSILONS;
```

Each member of the D3DXWELDEPSILONS structure gives the $\epsilon$ tolerance for the corresponding vertex component. The SkinWeights member gives the tolerance used for the vertex blend weights, Normal gives the tolerance for the surface normal coordinates and the Tex array gives the tolerances for each texture coordinate set. The Flags member contains a combination of one or more values from the D3DXWELDEPSILONSFLAGS enumeration.

```
enum _D3DXWELDEPSILONSFLAGS
{
    D3DXWELDEPSILONS_WELDALL             = 0x1,
    D3DXWELDEPSILONS_WELDPARTIALMATCHES  = 0x2,
    D3DXWELDEPSILONS_DONOTREMOVEVERTICES = 0x4
};
```

The D3DXWELDEPSILONS_WELDALL flag indicates that all vertices marked as overlapping by their adjacency will be welded into a single vertex. The D3DX-WELDEPSILONS_WELDPARTIALMATCHES flag indicates that vertices considered equal by the $\epsilon$ criteria in the structure will be modified so that components considered equal are made identical. If all components are identical, then one of the vertices will be removed unless the D3DXWELDEPSILONS_DONOTREMOVEVERTICES flag is used.

Welding the vertices of a mesh can only reduce the number of vertices in a mesh where they are shared between triangles. By comparison, D3DXSimplify-Mesh performs a mesh simplification algorithm over the mesh, removing vertices and modifying the number and topology of the triangles in the mesh while attempting to preserve the overall shape of the mesh. See section 19.2 for more on the use of `const` in the `attribute_weights` parameter to this function. The function attempts to reduce the mesh to a desired number of vertices or triangles, based on the `D3DXMESHSIMP` value in the `options` parameter.

```
HRESULT D3DXSimplifyMesh(ID3DXMesh *mesh,
            const DWORD *adjacency,
            const D3DXATTRIBUTEWEIGHTS *const attribute_weights,
            const float *vertex_weights,
            DWORD minima,
            DWORD options,
            ID3DXMesh **result);

enum _D3DXMESHSIMP
{
    D3DXMESHSIMP_VERTEX = 0x1,
    D3DXMESHSIMP_FACE   = 0x2
};
```

Mesh simplification is performed by removing shared vertices in an "edge collapse" operation, as illustrated in figure 19.4. The simplification process is controlled by a set of weights for the vertex components that may be applied equally to every vertex in the mesh, or combined with a set of weights for each vertex to identify vertices that are more important. The `adjacency` parameter gives the adjacency information for the input mesh as an array of three `DWORD`s per triangle. The `minima` parameter gives the desired number of vertices or triangles. It may be impossible for the mesh to be simplified to the desired `minima`, in which case the function succeeds because the paramter is only a desired minimum value.

The `vertex_weights` parameter specifies a pointer to an array of `float`s, one per vertex, that specify the relative importance of each vertex within the mesh. This array can be used to designate a subset of the mesh vertices are more important; the higher the value of the vertex weight, the more important the vertex is considered to the mesh. Vertices with lower weights will be removed during simplification before vertices with higher weights. If this parameter is `NULL`, then all vertices are weighted equally with a value of one. The vertex component weights are described by the `D3DXATTRIBUTEWEIGHTS` structure.

```
typedef struct _D3DXATTRIBUTEWEIGHTS
{
    float Position;
    float Boundary;
    float Normal;
    float Diffuse;
    float Specular;
    float Tex[8];
} D3DXATTRIBUTEWEIGHTS;
```

Rather than specify an $\epsilon$ comparison criteria for considering vertices equal as in `D3DXWeldVertices`, this structure specifies a floating-point value that identifies the relative importance of the vertex component during simplification. The higher the value, the more important the component is to the appearance of the object modelled by the mesh. If the `attribute_weights` parameter is `NULL`, then the function uses the following default values:

| | | | |
|---|---|---|---|
| `Position` | 1 | `Diffuse` | 0 |
| `Boundary` | 1 | `Specular` | 0 |
| `Normal` | 1 | `Tex[0...7]` | 0 |

This has the effect of preserving the shape and surface normals of the object at the expense of changes in texture coordinates and vertex color components. The `Boundary` member is used for vertices that are on the "edge" of the mesh, i.e. vertices that are part of an edge that has no adjacent triangle.

### 19.6.4 Adding Mesh Data

Just as D3DX provides several functions for reducing the amount of data in a mesh, it also provides functions for adding data to a mesh. To add new vertex components to a mesh, the `CloneMesh` and `CloneMeshFVF` methods should be used. If a mesh already contains a normal component, `D3DXComputeNormals` can be used to compute the vertex surface normals as an average of all the face normals for the faces containing the vertex.

```
HRESULT D3DXComputeNormals(ID3DXBaseMesh *mesh,
            const DWORD *adjacency);
```

The `D3DXTessellateNPatches` function will tessellate the triangles in a mesh according to the n-patch tessellation algorithm, described in subsection 5.15.2 on page 192, with a linear interpolation order for the position component. The `segments` parameter corresponds to `RSPatchSegments`.The `quadratic` parameter can be used to select between linear and quadratic interpolation order for the normal component. If this value is `TRUE`, then quadratic interpolation is used for the normal component. This function can be used as a fallback for situations where $N$-patch tessellation is not supported directly by the device. The tessellated mesh and its adjacency information are returned in the `result` and `adjacency_out` parameters. The `adjacency_out` parameter may be `NULL` if the adjacency information is not needed.

TODO: RS Patch Segments

```
HRESULT D3DXTessellateNPatches(ID3DXMesh *mesh,
            const DWORD *adjacency_in,
            float segments,
            BOOL quadratic,
            ID3DXMesh **result,
            ID3DXBuffer **adjacency_out);
```

The D3DXComputeTangent function will compute the tangent-space basis vectors for the source mesh and store them in the texture coordinate component of the vertices in the destination mesh. The source_stage parameter gives the index of the texture coordinate set that contains the source texture coordinates used to compute the tangent-space basis vectors. The u_stage and v_stage parameters give the index of the texture coordinate set that will recieve the $u$ and $v$ basis vectors, respectively. Either of these parameters may be set to D3DX_COMP_TANGENT_NONE if the corresponding basis vector is not required. The wrap parameter controls the wrapping of the texture coordinates in the $u$ and $v$ directions. If the value of wrap is FALSE, then no wrapping is performed. If the value is TRUE, then wrapping is performed. The destination mesh parameter should be a valid ID3DXMesh interface pointer for an existing mesh; D3DXComputeTangent does not create a new mesh for the result. The adjacency parameter returns the adjacency of the resulting mesh. It may be NULL if the adjacency information is not neeeded.

```
HRESULT D3DXComputeTangent(ID3DXMesh *source,
            DWORD source_stage,
            ID3DXMesh *destination,
            DWORD u_stage,
            DWORD v_stage,
            DWORD wrap,
            DWORD *adjacency);

#define D3DX_COMP_TANGENT_NONE 0xFFFFFFFF
```

## 19.6.5 Mesh Intersection Tests

The D3DXIntersect function tests the intersection of a ray with a mesh, while D3DXIntersectSubset tests the intersection with a specific subset of a mesh. In each case, the origin and direction of the ray are passed in and the results of the test are returned. For D3DXIntersectSubset, the attribute parameter indicates the subset used for the test. The hit parameter indicates whether or not any intersection was found.

```
HRESULT D3DXIntersect(ID3DXBaseMesh *mesh,
            const D3DXVECTOR3 *origin,
            const D3DXVECTOR3 *direction,
            BOOL *hit,
            DWORD *face,
            float *u,
            float *v,
            float *distance,
            ID3DXBuffer **hits,
            DWORD *count);
```

```
HRESULT D3DXIntersectSubset(ID3DXBaseMesh *mesh,
            DWORD attribute,
            const D3DXVECTOR3 *origin,
            const D3DXVECTOR3 *direction,
            BOOL *hit,
            DWORD *face,
            float *u,
            float *v,
            float *distance,
            ID3DXBuffer **hits,
            DWORD *count);
```

If an intersection was found, the `face`, `u`, `v`, and `distance` parameters will return information about the triangle closest to the origin of the ray. The `face` parameter returns the zero-based number of the intersected triangle. The `u` and `v` arguments return the barycentric coordinates of the intersection point within the triangle. The intersection coordinate $P$ can be computed from the vertex positions $P_1$, $P_2$, and $P_3$ of the triangle with the following formula:

$$P = P_1 + u(P_2 - P_1) + v(P_3 - P_1)$$

The `distance` parameter returns the distance from the origin of the ray to the intersection point.

The ray may have intersected more than one triangle in the mesh or subset. If all the intersections are desired, the `hits` parameter returns information about each intersection and the `count` parameter returns the number of intersections. If only the closest intersection to the ray origin is desired, the `hits` and `count` parameter may be `NULL`. The `ID3DXBuffer` returned by these functions contains an array of `D3DXINTERSECTINFO` structures. The `FaceIndex`, `U`, `V` and `Dist` members of the structure correspond to the `face`, `u`, `v` and `distance` arguments for the closest intersecting triangle.

```
typedef struct _D3DXINTERSECTINFO
{
    DWORD FaceIndex;
    float U;
    float V;
    float Dist;
} D3DXINTERSECTINFO, *LPD3DXINTERSECTINFO;
```

The intersection tests are performed in the model space of the mesh's vertices. If an intersection in world or view space is desired, transform the ray from world or view coordinates into model coordinates by using the inverse of the corresponding transformation matrices.

### 19.6.6 Mesh Conversion

A mesh may have so many vertices that `DWORD` indices are required, but the device may only support `WORD` indices. The problem can be solved by splitting a single ID3DXMesh object into multiple meshes, where each of the resulting meshes has fewer vertices so that WORD indices may be used. D3DXSplitMesh can be used to split a single mesh into multiple meshes; see section 19.2 for more about the use of const in the mesh parameter to this function. The input mesh and its adjacency information are used as the source data for creating one or more meshes, each with no more than max_vertices vertices and constructed with the given D3DXMESH options. To split a mesh into one or more meshes suitable for use with WORD indices, use a value of 65534[2] for max_vertices.

The number of meshes created is returned in the meshes parameter, while the remaining ID3DXBuffer parameters return the meshes themselves and the corresponding triangle adjacency, triangle remap, and vertex remap arrays. The remap arrays are slightly different in the case of D3DXSplitMesh. In most functions, the *i*th index in the remap array corresponds to the new location for a vertex or triangle of the *i*th vertex in the source mesh. With D3DXSplitMesh, the index corresponds to the number of the vertex or triangle in the resulting mesh and the value in the remap array gives the number of the original vertex or triangle in the source mesh.

```
HRESULT D3DXSplitMesh(ID3DXMesh *const mesh,
            const DWORD *adjacency_in,
            const DWORD max_vertices,
            const DWORD options,
            DWORD *meshes,
            ID3DXBuffer **result,
            ID3DXBuffer **adjacency_out,
            ID3DXBuffer **face_remap,
            ID3DXBuffer **vertex_remap);
```

The result parameter returns an array of ID3DXMesh interface pointers. The other three buffers return an array of DWORD arrays. To store an array of arrays in an ID3DXBuffer, the buffer first contains $n$ DWORD pointers, one for each of the resulting meshes, followed by the DWORD data itself. The following code shows how to obtain the vertex remap information for vertex number three of the second mesh.

```
DWORD num_meshes = 0;
ID3DXBuffer *meshes = 0;
ID3DXBuffer *adjacencies = 0;
ID3DXBuffer *new_faces = 0;
ID3DXBuffer *new_verts = 0;
THR(::D3DXSplitMesh(mesh, adjacency, 0xFFFF-1, D3DXMESH_MANAGED,
    &num_meshes, &meshes, &adjacencies, &new_faces, &new_verts));
const DWORD new_vertex =
    static_cast<DWORD **>(new_verts->GetBufferPointer())[2][3];
```

[2] 0xFFFF-1

Using the `rt::dx_buffer` helper class, the casting operators and the direct manipulation of the underlying `ID3DXBuffer` pointer can be eliminated.

```
DWORD num_meshes = 0;
rt::dx_buffer<ID3DXMesh *> meshes;
rt::dx_buffer<DWORD *> adjacencies;
rt::dx_buffer<DWORD *> new_faces;
rt::dx_buffer<DWORD *> new_verts;
THR(::D3DXSplitMesh(mesh, adjacency, 0xFFFF-1, D3DXMESH_MANAGED,
    &num_meshes, &meshes, &adjacencies, &new_faces, &new_verts));
const DWORD new_vertex = new_verts[2][3];
```

D3DX also provides functions for converting a mesh subset into one or more triangle strips. `D3DXConvertMeshSubsetToSingleStrip` generates a single triangle strip for the triangles in the subset, with disjoint strips of triangles connected by degenerate triangles. `D3DXConvertMeshSubsetToStrips` generates one or more triangle strips for the triangles in the subset. In each case, the `attribute` and `mesh` parameters describe the mesh subset to convert. The `result` is an index buffer containing the indices of the vertices in the triangle strips. The index buffer is created according to the flags specified in `options`, which may be one or more of the `D3DXMESH` flags for index buffers.

```
HRESULT D3DXConvertMeshSubsetToSingleStrip(ID3DXBaseMesh *mesh,
            DWORD attribute,
            DWORD options,
            IDirect3DIndexBuffer9 **result,
            DWORD *num_indices);

HRESULT D3DXConvertMeshSubsetToStrips(ID3DXBaseMesh *mesh,
            DWORD attribute,
            DWORD options,
            IDirect3DIndexBuffer9 **result,
            DWORD *num_indices,
            ID3DXBuffer **strip_lengths,
            DWORD *num_strips);
```

For a single strip, the `num_indices` parameter returns the number of indices stored in the index buffer. The corresponding number of triangles in the strip is one third the number of indices, including degenerate triangles connecting disjoint triangles.

For multiple strips, the `num_indices` parameter returns the number of indices stored in the index buffer for all the strips. The number of triangles in all the strips is one third the number of indices. The `num_strips` parameter returns the number of strips created from the subset. The `strip_lengths` returns an array of `DWORD`s, one per strip, giving the number of triangles in the strip.

### 19.6.7 Meshes in X Files

D3DX provides functions for loading and saving triangle meshes to the X file format. The file format and its associated interfaces are described in detail in chapter 21. The `D3DXLoadMeshFromX` function opens the X file given by the `filename` ANSI string argument and creates the `result` mesh object according to the `D3DXMESH` flags given in `options`. This function scans the X file for `Mesh` template nodes and creates a single mesh object that contains the union of all the nodes found in the file.

```
HRESULT D3DXLoadMeshFromX(LPSTR filename,
            DWORD options,
            IDirect3DDevice9 *device,
            ID3DXBuffer **adjacency,
            ID3DXBuffer **materials,
            DWORD *num_materials,
            ID3DXMesh **result);
```

`MeshMaterialList` and `Material` nodes are scanned and used to divide the mesh into subsets, one per material per mesh. The `materials` parameter returns an array of `D3DXMATERIAL` structures containing the contents of the `Material` nodes. D3DX allocates space for the texture filename strings in the `ID3DX-Buffer` returned by the function. Be sure to perform a deep copy of the strings if you release the returned buffer.

```
struct D3DXMATERIAL
{
    D3DMATERIAL9 MatD3D;
    LPSTR        pTextureFilename;
};
typedef struct D3DXMATERIAL *LPD3DXMATERIAL;
```

As with other mesh creation functions, if the adjacency or materials are not needed, then the corresponding parameters may be `NULL`.

The `D3DXLoadMeshFromXInMemory` and `D3DXLoadMeshFromXResource` functions are similar to `D3DXLoadMeshFromX` and load meshes from X files in memory or in a Win32 resource. The file in memory is given as a pointer to the file data and the size of the data.

```
HRESULT D3DXLoadMeshFromXInMemory(BYTE *data,
            DWORD size,
            DWORD options,
            IDirect3DDevice9 *device,
            ID3DXBuffer **adjacency,
            ID3DXBuffer **materials,
            DWORD *num_materials,
            ID3DXMesh **result);
```

An X file stored as a Win32 resource is identified by the `module` in which the resource is stored, the `name` of the resource and the `type` of the resource. `::GetModuleHandle` can be used to obtain the handle of the module containing the resource. The `MAKEINTRESOURCE` macro can be used to construct the `name` parameter for resources identified by an ordinal number.

```
HRESULT D3DXLoadMeshFromXResource(HMODULE module,
            LPCTSTR name,
            LPCTSTR type,
            DWORD options,
            IDirect3DDevice9 *device,
            ID3DXBuffer **adjacency,
            ID3DXBuffer **materials,
            DWORD *num_materials,
            ID3DXMesh **result);
```

For example, the following code snippet loads two meshes: one in a data resource named `IDR_DATA1`, and a second in a custom `meshes` resource named `foo.x`. Both are stored in the module corresponding to the executable image for the process.

```
CComPtr<ID3DXMesh> mesh1, mesh2;
HMODULE module = ::GetModuleHandle(NULL);
THR(::D3DXLoadMeshFromXResource(module,
    MAKEINTRESOURCE(IDR_DATA1), RT_RCDATA,
    D3DXMESH_SYSTEMMEM, device, NULL, NULL, NULL, &mesh1));
THR(::D3DXLoadMeshFromXResource(module,
    _T("foo.x"), _T("meshes"),
    D3DXMESH_SYSTEMMEM, device, NULL, NULL, NULL, &mesh2));
```

The `D3DXLoadMeshFromXof` function loads a mesh from an `IDirectXFile-Data` object. A data object is obtained by using the parser interfaces described in chapter 21 to examine the nodes individually within an X file. The `data` parameter must point to a `Mesh` node, otherwise the function will fail.

```
HRESULT D3DXLoadMeshFromXof(IDirectXFileData *data,
            DWORD options,
            IDirect3DDevice9 *device,
            ID3DXBuffer **adjacency,
            ID3DXBuffer **materials,
            DWORD *num_materials,
            ID3DXMesh **result);
```

The `D3DXSaveMeshToX` function can be used to save a mesh template and its associated materials to an X file; see section 19.2 for more about the use of `const` in the materials parameter to this function. The `mesh` and `adjacency` parameters give the mesh to be saved to the file. If you do not have the adjacency array for a mesh, you can obtain it with `GenerateAdjacency`, or you can pass `NULL`. If `NULL` is used, then no `VertexDuplicationIndices` node will appear in the X file. The `materials` and `num_materials` parameters describe the materials corresponding to the mesh. The array should be one larger than the largest attribute identifier in the attribute buffer for the mesh. So if the mesh has attribute identifiers 0, 10, 25, then the array of materials should hold 26 elements, numbered 0 through 25. Otherwise, the `MeshMaterialList` template will be written out incorrectly. If you have used a sparse set of attribute identifiers for your subsets, you can compact them into a contiguous set of zero-based identifiers by locking and modifying the attribute buffer.

```
HRESULT D3DXSaveMeshToX(LPSTR filename,
            ID3DXMesh *mesh,
            const DWORD *adjacency,
            const D3DXMATERIAL *const materials,
            DWORD num_materials,
            DWORD format);
```

## 19.7 Progressive Mesh

A progressive mesh allows for dynamic adjustments to the number of triangles and vertices used in the mesh. Mesh simplification can be an expensive operation and are generally used in non-interactive situations, while progressive meshes are intended to be used interactively. An instance of the progressive mesh interface can be created from an existing `ID3DXMesh` with `D3DXGeneratePMesh`; see section 19.2 for more about the use of `const` in the `attribute_weights` parameter. The `mesh` and `adjacency` parameters describe the input mesh. The `attribute_weights` and `vertex_weights` parameters describe the simplification criteria for the input mesh and are as described on page 691 for the `D3DXSimplifyMesh` function. The mesh is simplified to `min_value` triangles or vertices, depending on the `D3DXMESHSIMP` value in the `options` parameter.

```
HRESULT D3DXGeneratePMesh(ID3DXMesh *mesh,
            const DWORD *adjacency,
            const D3DXATTRIBUTEWEIGHTS *const attribute_weights,
            const float *vertex_weights,
            DWORD min_value,
            DWORD options,
            ID3DXPMesh **result);
```

A progressive mesh can also be created from a COM stream interface in which the X file representation of the progressive mesh was previously written. A COM stream interface is an I/O abstraction and is described in detail in the MSDN documentation and books describing COM structured storage.

```
HRESULT D3DXCreatePMeshFromStream(IStream *stream,
            DWORD options,
            IDirect3DDevice9 *device,
            ID3DXBuffer **materials,
            DWORD *num_materials,
            ID3DXPMesh **result);
```

As an example, the following code opens a COM structured storage on the file `tmp.pm`, obtains the stream interface for the stream named `CONTENTS`. The stream is used to create the progressive mesh in system memory, ignoring any mesh materials. In this example, a file was used, but a stream can also be created on a chunk of memory allowing you to create a progressive mesh from a resource by obtaining a pointer to the resource's data in memory. It is also possible to implement your own stream object on any internal data source, such as a network connection.

```
CComPtr<IStorage> storage;
THR(::StgOpenStorageEx(L"tmp.pm",
    STGM_READ | STGM_SHARE_EXCLUSIVE | STGM_DIRECT,
    STGFMT_DOCFILE, 0, NULL, NULL, IID_IStorage,
    reinterpret_cast<void **>(&storage)));
CComPtr<IStream> stream;
THR(storage->OpenStream(L"CONTENTS", NULL,
    STGM_READ | STGM_SHARE_EXCLUSIVE | STGM_DIRECT, 0,
    &stream));
CComPtr<ID3DXPMesh> pmesh;
THR(::D3DXCreatePMeshFromStream(stream, D3DXMESH_SYSTEMMEM,
    m_pd3dDevice, NULL, NULL, &pmesh));
```

The progressive mesh interface is summarized in interface 19.3. `ID3DXPMesh` derives from `ID3DXBaseMesh`, described in section 19.5. The `GetMinFaces`, `GetMaxFaces`, `GetMinVertices` and `GetMaxVertices` methods describe the range of dynamic simplification provided by the progressive mesh. To change any of these values, the mesh must be destroyed and recreated with the new desired minimum and maximum. The progressive mesh uses the methods of `ID3DXBaseMesh` to expose any number of triangles and vertices within the range described by its read-only properties. When the progressive mesh is initially created, it uses the minimum number of its vertices and triangles.

Interface 19.3: Summary of the `ID3DXPMesh` interface.

**ID3DXPMesh**

| **Read-Only Properties** | |
|---|---|
| `GetAdjacency` | Triangle adjacency information |
| `GetMaxFaces` | Maximum number of triangles |
| `GetMinFaces` | Minimum number of triangles |
| `GetMaxVertices` | Maximum number of vertices |
| `GetMinVertices` | Minimum number of vertices |

| **Write-Only Properties** | |
|---|---|
| `SetNumFaces` | Set the level of detail as the number of triangles |
| `SetNumVertices` | Set the level of detail as the number of vertices |

| **Methods** | |
|---|---|
| `ClonePMeshFVF` | Make a copy of the mesh with an FVF code |
| `ClonePMesh` | Make a copy of the mesh |
| `Optimize` | Make an optimized copy of the mesh |
| `OptimizeBaseLOD` | Optimize the base level of detail of the mesh |
| `Save` | Write the progressive mesh to a COM stream |
| `TrimByFaces` | Change the level of detail range as a number of triangles |
| `TrimByVertices` | Change the level of detail range as a number of vertices |

```
interface ID3DXPMesh : ID3DXBaseMesh
{
    // read-only properties
    HRESULT GetAdjacency(DWORD *adjacency);
    DWORD GetMaxFaces();
    DWORD GetMaxVertices();
    DWORD GetMinFaces();
    DWORD GetMinVertices();

    // write-only properties
    HRESULT SetNumFaces(DWORD value);
    HRESULT SetNumVertices(DWORD value);

    // methods
    HRESULT ClonePMesh(DWORD options,
                const DWORD *declaration,
                IDirect3DDevice9 *device,
                ID3DXPMesh **resuilt);
    HRESULT ClonePMeshFVF(DWORD options,
                DWORD fvf,
                IDirect3DDevice9 *device,
                ID3DXPMesh **result);
    HRESULT Optimize(DWORD flags,
                DWORD *adjacency_out,
                DWORD *face_remap,
                ID3DXBuffer **vertex_remap,
                ID3DXMesh **result);
    HRESULT OptimizeBaseLOD(DWORD flags,
                DWORD *face_remap);
    HRESULT Save(IStream *stream,
                D3DXMATERIAL *materials,
                DWORD num_materials);
    HRESULT TrimByFaces(DWORD minima,
                DWORD maxima,
                DWORD *face_remap,
                DWORD *vertex_remap);
    HRESULT TrimByVertices(DWORD minima,
                DWORD maxima,
                DWORD *face_remap,
                DWORD *vertex_remap);
};
```

The `GetAdjacency` method returns the adjacency array for the current number of triangles in the mesh, but the size of the array must be large enough to hold the adjacency information for the maximum number of triangles in the mesh. The entire array contents will be overwritten, but only the triangles in the current level of detail will contain useful information. For instance, if the maximum number of triangles in the mesh is 100 and the current level of detail is 50 faces, then the adjacency array passed to `GetAdjacency` must be at least 300 `DWORD`s in size and the first 150 `DWORD`s will contain the adjacency for the 50 triangles in the current level of detail and the remaining `DWORD`s will contain `0xFFFFFFFF`.

The write-only properties of the mesh are used to set the number of active triangles and vertices. The range of allowed values is restricted to the minimum and maximum number of triangles or vertices, as reported by the read-only properties of the mesh. The number of triangles after this call may be off by one since an edge collapse transformation may remove one or two triangles, as shown in figure 19.4.

The `ClonePMesh` and `ClonePMeshFVF` create a copy of the entire progressive mesh, using the new vertex declaration or FVF code, respectively, for the vertices of the new progressive mesh. This is in contrast to the `CloneMesh` and `CloneMeshFVF` methods of the base interface `ID3DXBaseMesh` which would only clone the set of active vertices and triangles.

The `Optimize` method performs similarly to the `Optimize` method of `ID3DX-Mesh`. The `OptimizeBaseLOD` optimizes the mesh for the number of triangles currently selected as the base level of detail for the mesh. The `flags` argument is one or more `D3DXMESHOPT` flags indicating how the mesh should be optimized. Indices are moved within the index buffer so that the currently selected triangles can be rendered as fast as possible. The `face_remap` parameter describes how the triangles were moved within the index buffer and consists of one `DWORD` per triangle, giving the new position for each triangle. The remap array should be large enough to contain one `DWORD` for the maximum number of triangles in the mesh.

The `TrimByFaces` and `TrimByVertices` methods adjust the level of detail range for the mesh. In each case the `minima` and `maxima` parameters give the new range for the number of triangles or vertices in the progressive mesh, respectively. The new values must be within the current minimum and maximum values. This function can only restrict the existing range of values to a tighter range, it cannot broaden the range. The `face_remap` and `vertex_remap` parameters return information on how the triangles and vertices were rearranged within the index and vertex buffers of the mesh. Each of these parameters should point to an array of `DWORD`s, one per triangle or vertex for the maximum number of each in the mesh.

The `Save` method writes the progressive mesh data structures to a COM stream interface as a binary X file. The progressive mesh can be reconstituted from this data with the `D3DXCreatePMeshFromStream` function. The following code shows how to write a progressive mesh to a COM structured storage.

```
CComPtr<IStream> stream;
CComPtr<IStorage> storage;
THR(::StgCreateStorageEx(L"tmp.pm",
    STGM_READWRITE | STGM_SHARE_EXCLUSIVE | STGM_CREATE |
    STGM_DIRECT, STGFMT_DOCFILE, 0, NULL, 0,
    IID_IStorage, reinterpret_cast<void **>(&storage)));
THR(storage->CreateStream(L"CONTENTS",
    STGM_READWRITE | STGM_SHARE_EXCLUSIVE | STGM_CREATE |
    STGM_DIRECT, 0, 0, &stream));
THR(pmesh->Save(stream, NULL, 0));
```

## 19.8 Simplification Mesh

The simplification mesh interface is used as a factory for creating plain and progressive meshes and is intended for use in offline tools or during initialization. The `ID3DXSPMesh` interface derives directly from `IUnknown` and provides no means for rendering its mesh data. An instance of this interface is created with the `D3DXCreateSPMesh` function; see section 19.2 for more about the use of `const` in the `attribute_weights` parameter. The `mesh` and `adjacency` parameters give the input mesh that is the source of the vertex data used by `ID3DXSPMesh`. The `attribute_weights` and `vertex_weights` parameters give the simplification criteria used to simplify the mesh, as described on page 691 for `D3DXSimplifyMesh`. The interface is summarized in interface 19.4.

```
HRESULT D3DXCreateSPMesh(ID3DXMesh *mesh,
            const DWORD *adjacency,
            const D3DXATTRIBUTEWEIGHTS *const attribute_weights,
            const float *vertex_weights,
            ID3DXSPMesh **result);
```

Interface 19.4: Summary of the `ID3DXSPMesh` interface.

**ID3DXSPMesh**

| **Read-Only Properties** | |
|---|---|
| `GetDeclaration` | The vertex shader declaration for the mesh vertices |
| `GetDevice` | The associated device |
| `GetFVF` | The FVF code for the mesh vertices |
| `GetMaxFaces` | The maximum number of triangles in the simplification mesh |
| `GetMaxVertices` | The maximum number of vertices in the simplification mesh |
| `GetNumFaces` | The number of triangles in the simplification mesh |
| `GetNumVertices` | The number of vertices in the simplification mesh |
| `GetOptions` | The mesh options for the simplification mesh when it was created |
| `GetVertexAttributeWeights` | The per-vertex attribute simplification weights |
| `GetVertexWeights` | The per-vertex simplification weights |

| **Methods** | |
|---|---|
| `CloneMesh` | Clone the mesh using a vertex shader declarator |
| `CloneMeshFVF` | Clone the mesh using an FVF code |
| `ClonePMesh` | Clone a progressive mesh using a vertex shader declarator |
| `ClonePMeshFVF` | Clone a progressive mesh using an FVF code |
| `ReduceFaces` | Reduce the number of triangles |
| `ReduceVertices` | Reduce the number of vertices |

```
interface ID3DXSPMesh : IUnknown
{
    // read-only properties
    HRESULT GetDeclaration(DWORD value[MAX_FVF_DECL_SIZE]);
    HRESULT GetDevice(IDirect3DDevice9 **value);
    DWORD GetFVF();
    DWORD GetMaxFaces();
    DWORD GetMaxVertices();
    DWORD GetNumFaces();
    DWORD GetNumVertices();
    DWORD GetOptions();
    HRESULT GetVertexAttributeWeights(
                D3DXATTRIBUTEWEIGHTS *value);
    HRESULT GetVertexWeights(float *value);

    // methods
    HRESULT CloneMesh(DWORD options,
                const DWORD *declaration,
                IDirect3DDevice9 *device,
                DWORD *adjacency,
                DWORD *vertex_remap,
                ID3DXMesh **result);
    HRESULT CloneMeshFVF(DWORD options,
                DWORD fvf,
                IDirect3DDevice9 *device,
                DWORD *adjacency,
                DWORD *vertex_remap,
                ID3DXMesh **result);
    HRESULT ClonePMesh(DWORD options,
                const DWORD *declaration,
                IDirect3DDevice9 *device,
                DWORD *vertex_remap,
                ID3DXPMesh **result);
    HRESULT ClonePMeshFVF(DWORD options,
                DWORD fvf,
                IDirect3DDevice9 *device,
                DWORD *vertex_remap,
                ID3DXPMesh **result);
    HRESULT ReduceFaces(DWORD faces);
    HRESULT ReduceVertices(DWORD vertices);
};
```

The `GetDevice` method returns the device associated with the mesh. The vertex declaration and FVF code are returned by the `GetDeclaration` and `GetFVF` methods, respectively. The mesh only handles vertices that can be described by FVF codes and the declaration property is just a convenience. The `D3DXMESHOPT` flags for the mesh are returned by the `GetOptions` method and are identical to the `D3DXMESHOPT` flags used to create the source mesh.

The maximum number of triangles and vertices are given by the `GetMaxFaces` and `GetMaxVertices` methods, respectively. This number can be reduced through mesh simplification, and the number of triangles and vertices currently set on the mesh are returned by the `GetNumFaces` and `GetNumVertices` methods, respectively.

The `GetVertexAttributeWeights` method returns the simplification weights of the vertex components in a `D3DXATTRIBUTEWEIGHTS` structure. The `GetVertexWeights` method returns the simplification weights associated with each vertex. The `value` parameter should point to an array of `float`s, one for each vertex in the original mesh.

The CloneMesh and CloneMeshFVF methods create an ID3DXMesh interface corresponding to the currently selected number of triangles and vertices in the simplification mesh. The new mesh is created with the D3DXMESH flags in the options parameter and associated with the device given in the device parameter. The adjacency parameter returns the adjacency of the created mesh. The vertex_remap parameter returns an array of DWORDs, one for each vertex in the original mesh, describing how the original vertices were copied into the vertex buffer of the clone. The ClonePMesh and ClonePMeshFVF methods function similarly to the CloneMesh and CloneMeshFVF methods, but create the clone as an ID3DXPMesh instead of an ID3DXMesh.

The ReduceFaces and ReduceVertices methods simplify the mesh. Once the number of faces or vertices has been reduced, it cannot be increased again. Plain meshes cloned from a simplification mesh with a reduced face or vertex count will also have a correspondingly reduced face or vertex count. Progressive meshes cloned from a simplification mesh with a reduced face or vertex count will have a minimum face or vertex count that corresponds to the reduced count and a maximum face or vertex count corresponding to the original counts.

## 19.9 Skin Mesh

D3DX provides the ID3DXSkinInfo interface as a factory for creating meshes that contain vertex blending weights and associated "bones". Each bone in the mesh describes a mapping between the model coordinate space of the vertex positions and the world coordinate space, as described in chapter 5. A skin mesh may be created with up to 256 bones and a vertex may be influenced by an arbitrary subset of the bones. The influence, or weighting, of each bone on a vertex within the mesh is given by a floating-point value. All the bone influences for any given vertex must sum to one.

The skin mesh can't be drawn directly from its interface. It must first be converted to an ID3DXMesh that uses vertex blending or indexed vertex blending. During conversion, a skin mesh attempts to map its arbitrary number of bones and vertex influences to the capabilities of the device. In the typical case, each vertex within the mesh is affected by only a small number of bones. Such a mesh is converted by splitting the mesh into subsets where each of the triangles within the subset fit within the vertex blending capabilities of the device. For cases where a vertex is influenced by more bones than is supported by a device, bones with the smallest influence on a vertex are dropped until the triangle containing the vertex can be rendered.

An ID3DXMesh can also be generated that does not use vertex blending. In this case, the resulting mesh vertices have been transformed and blended into the pose defined by the bone influences currently set on the skin mesh. The pose can be changed by either using the same bone transformations and changing the bone influences on the vertices or changing the bone transformation matrices. The generated ID3DXMesh can then be updated with the vertices of the new pose.

An empty skin mesh can be created by describing the vertex format with a declaration or an FVF code with the `D3DXCreateSkinInfo` and `D3DXCreateSkinInfoFVF` functions, respectively. In each case, the `num_faces` parameter gives the number of triangles that will be needed for the mesh, the `options` parameter gives one or more `D3DXMESH` flags used for the mesh's index and vertex buffers created on the associated `device`. The `num_bones` parameter gives the total number of unique bones used by the mesh.

```
HRESULT D3DXCreateSkinMesh(DWORD num_faces,
            DWORD num_vertices,
            DWORD num_bones,
            DWORD options,
            const DWORD *declaration,
            IDirect3DDevice9 *device,
            ID3DXSkinMesh **result);

HRESULT D3DXCreateSkinMeshFVF(DWORD num_faces,
            DWORD num_vertices,
            DWORD num_bones,
            DWORD options,
            DWORD fvf,
            IDirect3DDevice9 *device,
            ID3DXSkinMesh **result);
```

A populated skin mesh can be created from an existing mesh with the `D3DXCreateSkinInfoFromBlendedMesh` function. The vertex, index and attribute buffers of the source mesh are shared with the skin mesh. Modifying the buffer contents of either mesh affects both meshes. Space for `num_bones` bones is allocated in the skin mesh, but the application must still fill in the bone influence weights for each bone.

```
HRESULT D3DXCreateSkinMeshFromMesh(ID3DXMesh *mesh,
            DWORD num_bones,
            ID3DXSkinMesh **result);
```

All three functions return an `ID3DXSkinInfo` interface, which derives directly from `IUnknown`. The interface is summarized in interface 19.5.

Interface 19.5: Summary of the `ID3DXSkinInfo` interface.

**ID3DXSkinInfo**

| **Read-Only Properties** | |
|---|---|
| `GetMaxFaceInfluences` | ... |
| `GetMaxVertexInfluences` | ... |
| `GetNumBoneInfluences` | ... |

| | |
|---|---|
| `GetNumBones` | ... |

| **Properties** | |
|---|---|
| `GetBoneInfluence` | ... |
| `SetBoneInfluence` | |
| `GetBoneName` | ... |
| `SetBoneName` | |
| `GetBoneOffsetMatrix` | ... |
| `SetBoneOffsetMatrix` | |
| `GetBoneVertexInfluence` | ... |
| `SetBoneVertexInfluence` | |
| `GetDeclaration` | ... |
| `SetDeclaration` | |
| `GetFVF` | ... |
| `SetFVF` | |
| `GetMinBoneInfluence` | ... |
| `SetMinBoneInfluence` | |

| **Methods** | |
|---|---|
| `Clone` | Clone the skin information |
| `ConvertToBlendedMesh` | Copy to a blended mesh |
| `ConvertToIndexed-BlendedMesh` | Copy to an indexed blended mesh |
| `FindBoneVertex-InfluenceIndex` | ... |
| `Remap` | ... |
| `UpdateSkinnedMesh` | Update the bone information |

```
interface ID3DXSkinMesh : IUnknown
{
    // read-only properties
    HRESULT GetDeclaration(DWORD value[MAX_FVF_DECL_SIZE]);
    HRESULT GetDevice(IDirect3DDevice9 **value);
    DWORD GetFVF();
    HRESULT GetIndexBuffer(IDirect3DIndexBuffer9 **value);
    HRESULT GetMaxFaceInfluences(DWORD *value);
    HRESULT GetMaxVertexInfluences(DWORD *value);
    DWORD GetNumBoneInfluences(DWORD bone);
    DWORD GetNumBones();
    DWORD GetNumFaces();
    DWORD GetNumVertices();
    DWORD GetOptions();
    HRESULT GetOriginalMesh(ID3DXMesh **value);
    HRESULT GetVertexBuffer(IDirect3DVertexBuffer9 **value);
    // read-write properties
    HRESULT GetBoneInfluence(DWORD bone,
                DWORD *vertices,
                float *value);
    HRESULT SetBoneInfluence(DWORD bone,
                DWORD num_influences,
                const DWORD *vertices,
                const float *value);

    // methods
    HRESULT ConvertToBlendedMesh(DWORD options,
                const DWORD *const adjacency_in,
                DWORD *adjacency_out,
                DWORD *num_bone_combinations,
                ID3DXBuffer **bone_combinations,
                DWORD *face_remap,
                ID3DXBuffer **vertex_remap,
                ID3DXMesh **result);
    HRESULT ConvertToIndexedBlendedMesh(DWORD options,
                const DWORD *const adjacency_in,
                DWORD palette_size,
                DWORD *adjacency_out,
                DWORD *num_bone_combinations,
                ID3DXBuffer **bone_combinations,
                DWORD *face_remap,
                ID3DXBuffer **vertex_remap,
                ID3DXMesh **result);
    HRESULT GenerateSkinnedMesh(DWORD options,
                float min_weight,
                const DWORD *const adjacency_in,
                DWORD *adjacency_out,
                DWORD *face_remap,
                ID3DXBuffer **vertex_remap,
                ID3DXMesh **result);
    HRESULT LockAttributeBuffer(DWORD flags, DWORD **data);
    HRESULT LockIndexBuffer(DWORD flags, BYTE **data);
    HRESULT LockVertexBuffer(DWORD flags, BYTE **data);
    HRESULT UnlockAttributeBuffer();
    HRESULT UnlockIndexBuffer();
    HRESULT UnlockVertexBuffer();
    HRESULT UpdateSkinnedMesh(const D3DXMATRIX *bone_transforms,
                const D3DXMATRIX *bone_inverse_transforms,
                ID3DXMesh *mesh);
};
```

TODO: xsiGetOptions

The format of the vertices in the mesh are returned by the `GetDeclaration` and GetFVF methods. Skin meshes only support vertices in a format compatible with FVF codes. The D3DXMESH flags used to create the skin mesh are returned by the xsiGetOptions method.

TODO: xsiGetNumFaces, xsiGetNumVertices, xsiGetIndexBuffer, xsiGetVertexBuffer, GetOriginalMesh

The number of triangles and vertices in the skin mesh are returned by the xsiGetNumFaces and xsiGetNumVertices methods. The underlying index and vertex buffers can be directly manipulated by obtaining their interface pointers with the xsiGetIndexBuffer and xsiGetVertexBuffer methods. A copy of the vertices and indices stored in the skin mesh can be obtained with the GetOriginalMesh method. The vertices will not contain blend weights, nor will they have been transformed by any bone influences. The resulting mesh is called the "original pose" of the skin mesh.

TODO: xsiLockVertexBuffer, xsiUnlockVertexBuffer, xsiLockIndexBuffer, xsiUnlockIndexBuffer, xsiLockAttributeBuffer, xsiUnlockAttributeBuffer

The vertex, index and attribute data defining the original pose of the skin mesh can be accessed directly with the xsiLockVertexBuffer, xsiUnlockVertexBuffer, xsiLockIndexBuffer, xsiUnlockIndexBuffer, xsiLockAttributeBuffer, and xsiUnlockAttributeBuffer methods. The usage of these methods is identical to that of ID3DXMesh and ID3DXBaseMesh. Creating helper locking classes for the buffers of a skin mesh will ensure that matched pairs of lock and unlock methods are always called. Rather than create separate locking helper classes for each of the ID3DXBaseMesh, ID3DXMesh and ID3DXSkinInfo interfaces, we can use a template parameter in the helper class to specify the base interface for the lock. The helper classes in the sample code use this technique.

As with vertex blending, described in section 6.7, each vertex in a skin mesh is influenced by a number of bones. Each bone influence on a vertex is described by a floating-point weight that describes how much of an influence the corresponding bone transformation has on the final position of this vertex.

The number of bones in the skin mesh is returned by the GetNumBones method. The number of vertices in the mesh influenced by a bone is returned by the GetNumBoneInfluences method when the bone parameter is a valid bone number, or zero otherwise. The maximum number of bones influencing any vertex in the mesh is returned by GetMaxVertexInfluences.

The maximum number of bones influencing any of the triangles in the mesh is returned by the GetMaxFaceInfluences method. This value is used in conjunction with the D3DXBONECOMBINATION structure returned by the conversion methods. This is determined by looking at the bones influencing each of the three vertices in each triangle in the mesh. If the maximum number of bones influencing a triangle in the mesh is smaller than the MaxVertexBlendMatrices member of D3DCAPS9, then the mesh can be rendered as a blended mesh using all the defined bones. If the number is larger, then the mesh may be rendered as an indexed blended mesh, depending on the device support for indexed vertex blending.

The GetBoneInfluence and SetBoneInfluence methods expose the influence each bone has on the vertices of the skin mesh as a read/write property. To set the bone influences, pass the bone number, the number of influences, an array of DWORDs, one per influence, giving the vertices affected by the bone, and an array of floats, one per influence, giving the blend weight for each vertex. When obtaining the bone influences, the arrays passed in the weights and `vertices` parameters must be large enough to hold one value for each vertex in the skin mesh.

For example, the following code sets the influences for bone one on vertices 15 and 20 to 1.0 and 0.5, respectively.

```
const float weights[2] =
{
    1.0f, 0.5f
};
const DWORD vertices[2] =
{
    15, 20
};
THR(skin_mesh->SetBoneInfluence(1, 2, &weights[0], &vertices[0]));
```

This code obtains the influences for bone zero of the skin mesh. The C++ standard vector class is used to allocate the appropriate storage for the vertex numbers and their associated floating-point weights.

```
std::vector<float> weights(skin_mesh->GetNumVertices());
std::vector<DWORD> vertices(skin_mesh->GetNumVertices());
THR(skin_mesh->GetBoneInfluence(0, &vertices[0], &weights[0]));
```

Once a skin mesh has been loaded with vertex data and all the bone influences for all the vertices have been set, it can be converted to an `ID3DXMesh` for rendering. The `ConvertToBlendedMesh` method creates a mesh that uses non-indexed vertex blending for the skinning, while `ConvertToIndexedBlendedMesh` uses indexed vertex blending. For the indexed case, the `palette_size` parameter gives the number of vertex blend matrices to use in the indexed blended mesh. The remaining parameters to each of these functions are treated identically. See section 19.2 for more on the use of `const` in the parameters to these methods.

The resulting mesh is created according to the `D3DXMESH` flags in the `options` parameter. Adjacency information for the skin mesh is passed to the method and the adjacency information for the created mesh is returned to the caller. The face and vertex remap arrays return `DWORD`s identifying how the triangles and vertices in the skin mesh were mapped onto the triangles and vertices of the created mesh. These arrays have one `DWORD` for each triangle or vertex in the created mesh.

The `num_bone_combinations` parameter returns the number of `D3DXBONE-COMBINATION` structures returned in the `bone_combinations` array. Each structure describes how the skin mesh was subdivided to accommodate the blending limitations of the device associated with the skin mesh.

```
typedef struct _D3DXBONECOMBINATION
{
    DWORD AttribId;
    DWORD FaceStart;
    DWORD FaceCount;
    DWORD VertexStart;
    DWORD VertexCount;
    DWORD *BoneId;
} D3DXBONECOMBINATION, *LPD3DXBONECOMBINATION;
```

The AttribId member gives the attribute ID associated with this bone combination. The face and vertex members describe the range of triangles and vertices in the created mesh governed by this combination of bones. The BoneId member points to an array of DWORDs giving the bones influencing this range of triangles. The array size is the number of maximum bone influences for any face in the skin mesh.

To render the resulting ID3DXMesh, the bone combination structure indicates the bones used for each blended piece. The matrix for the first bone listed in the structure should be set as the first world matrix, the matrix for the second bone listed in the structure should be set as the second world matrix, and so-on.

TODO: GenerateSkinnedMesh

Another approach is to perform the vertex blending entirely on the CPU and draw a plain mesh after the blending has been performed. For this approach, the application first calls GenerateSkinnedMesh to create a mesh in the initial pose. See section 19.2 for more on the use of const with the adjacency parameter to this method. The min_weight parameter gives a minimum weight below which bone influences for any particular vertex will be ignored.

The mesh is created according to the D3DXMESH flags specified in the options parameter. The adjacency of the skin mesh is passed into the method with the adjacency_in parameter and the adjacency of the created ID3DXMesh is returned in the adjacency_out parameter. The face_remap and vertex_remap parameters return arrays defining the mapping of triangles and vertices from the skin mesh into the created mesh. If the adjacency or remap information for the created mesh are not required, the parameters may be NULL.

To change the pose, the application calls UpdateSkinnedMesh with the mesh created by the generate method and the matrices for each bone in the mesh. The initial pose uses the identity matrix for all bones. The application passes two arrays of D3DXMATRIX structures, each array containing one matrix per bone. The first array is the forward transformation matrices that map model coordinates to bone coordinates and the second array contains the inverse matrices of the first array. When the method returns, the mesh parameter has had its vertices updated to reflect the new transformation matrices for each bone.

## 19.9.1 Skin Meshes in X Files

D3DX provides a function for reading skinned meshes from X files. This function reads the mesh data and the information contained in the XSkinMeshHeader and SkinWeights templates to obtain the necessary bone and vertex weight data.

```
HRESULT D3DXLoadSkinMeshFromXof(IDirectXFileData *data,
            DWORD options,
            IDirect3DDevice9 *device,
            ID3DXBuffer **adjacency,
            ID3DXBuffer **materials,
            DWORD *num_materials,
            ID3DXBuffer **bone_names,
            ID3DXBuffer **bone_transforms,
            ID3DXSkinMesh **result);
```

The `data` parameter should be an X file node containing a `Mesh` template. The `options` parameter contains a combination of `D3DXMESH` flags that describe the skin mesh's underlying vertex and index buffers created on the `device`. The `adjacency` parameter returns an `ID3DXBuffer` containing the adjacency information for the mesh, as an array of three `DWORD`s for each triangle in the mesh. The `materials` parameter is an `ID3DXBuffer` containing an array of `D3DX-MATERIAL` structures. The size of the array is returned in the `num_materials` parameter. The names of the bones are returned in the `bone_names` parameter as an `ID3DXBuffer` containing an array of ANSI strings, one for each bone in the skin mesh. Similarly, the `bone_transforms` parameter returns the bone transformations as an `ID3DXBuffer` containing an array of `D3DXMATRIX` structures, one for each bone in the skin mesh.

## 19.10 `rt_D3DXSphere` Sample Application

The `rt_D3DXSphere` sample application demonstrates the use of the D3DX primitive construction functions `D3DXCreatePolygon`, `D3DXCreateBox`, `D3DXCreate-Cylinder`, `D3DXCreateSphere`, `D3DXCreateTorus` and `D3DXCreateTeapot`. You can interactively adjust the parameters for these shapes from the keyboard. The source to the application is located in the sample code accompanying this book.

## 19.11 SDK Samples

Almost every sample in the SDK uses a D3DX mesh object for loading models and scenes. The following samples demonstrate mesh functionality specifically, as opposed to using a mesh object to demonstrate another feature.

**EnhancedMesh** The enhanced mesh sample demonstrates $N$-Patch tessellation on a mesh. You can load an arbitrary X file into the sample from the menu.

**OptimizedMesh** The optimized mesh sample demonstrates the improvement in rendering performance by using the `Optimize` method to reorder triangles and vertices. You can select the mesh optimization options and load an arbitrary X file into the sample from the menu.

**ProgressiveMesh** The progressive mesh sample demonstrates the dynamic level of detail feature provided by `ID3DXPMesh`. The number of vertices in the mesh can be changed with the keyboard and an arbitrary X file can be loaded into the sample from the menu. The `knot.x` file in the SDK works particularly well with this sample, although generating the initial progressive mesh can take some time, even on a fast CPU.

**SkinnedMesh** The skinned mesh sample is the most advanced mesh sample in the SDK. This sample demonstrates how to parse an X file node by node so that the coordinate frame hierarchy stored in the file can be replicated by program data structures. It also parses animation information from the X file to create an animated rendering. The skin mesh information is read from the file and an `ID3DXSkinInfo` is constructed. The skin mesh is used to create meshes for rendering the model using one of the following four methods.

1. non-indexed vertex blending
2. indexed vertex blending
3. software skinning
4. vertex shader based indexed skinning

In addition to the samples in the SDK that use meshes, tutorial 6 in the SDK documentation demonstates the basics of the `ID3DXMesh` object in the context of a very simple program.

## 19.12 Further Reading

Research in meshing algorithms have received a great deal of attention in the past few years. Work in this area continues to be published in the proceedings of the SIGGRAPH conference, among others. The mesh simplification used in D3DX is based on the Garland-Heckbert quadric error metric, with refinements by Hugues Hoppe to accommodate normal and attribute space metrics.

*Surface Simplification Using Quadric Error Metrics*, by Michael Garland and Paul S. Heckbert. SIGGRAPH 1997 Conference Proceedings, Annual Conference Series, pp. 209-216, Addison Wesley, August 1997.

Hugues Hoppe's home page contains links for his papers on meshes.
`http://research.microsoft.com/~hoppe/`

*The Origins of the Teapot*, IEEE Computer Graphics and Applications, 7(1), pp. 8-19, January 1987.