Part I
ADVANCED TOPICS

- ★ **Chapter 1:** **3D Models, Meshes, and Tween Meshes**
- ★ **Chapter 2:** **Working with Input Devices**
- ★ **Chapter 3:** **Building a GUI System**
- ★ **Chapter 4:** **Using the Stencil Buffer**
- ★ **Chapter 5:** **Shaders and Effect Files**

Chapter 1

Models, Meshes, and Tween Meshes

- **loading and rendering model files**
- **.X and .ac3d file format**
- **concept of vertex tweening**
- **hardware and software tweening**
- **animating a model by tweening**

"There is no strength in numbers have no such misconception"
(Uriah Heep, Lady in Black)

1.1 Basics about Models

In the last chapters you have seen how you can define a simple rectangle or cube by hard-coding. That means that you will just define the necessary vertices in the source code by writing down the positions of the vertices as well as their normal vectors, their texture coordinates, and all other attributes.

This is nice for doing some tests of the render functionality but of course that is not the way to go for more complex models. Just imagine writing down the attributes for some ten thousands of vertices you would need for a nice and detailed model. In this chapter you will learn how to use prefabricated models that you can load as meshes into your application from a file.

1.1.1 Meshes versus Models

Basically the terms mesh and model refer more or less to the same thing. In general the term mesh should rather be polymesh because it describes an object that is build from polygons. Actually a mesh is nothing more than just a bunch of polygons in 3D space which belong to the same group. The term model is used to describe a three-dimensional, virtual representation of real world's or a fantasy world's object.

But after all a model is in fact represented as a mesh in computer graphics. There are some slight differences between those two terms which also depend on the guys you talk to about them for that matter. One very important difference is that a model can contain several meshes where each mesh represents a certain part of the whole model. A model of a battle tank can for example have a mesh for the chassis, a mesh for the wheels and the track, as well as a mesh for the turret. Such a design has its advantages. You could then controls each mesh of the model separately to rotate the turret without rotating the chassis for instance.

But for the remainder of this book the terms mesh and model will be used more or less exchangeable. When talking about a model I will normally refer to the file containing the data and when talking about a mesh I will normally mean the data that is loaded into a certain data structure in the engine or in an application.

1.1.2 Building Models

Now everyone is talking about cool 3D models and in this chapter we will also start using them by loading model files into data structures. But the big question is where those models actually come from and how they made their way into the file in the first place. To give truth the honor I have to say that those models will normally come from a 3D artist who is a talented guy with regard to building models. And he has tools to support him as well.

If you are a 3D graphics programmer reading this then be warned. A lot of peopled from our side tried to build 3D models. But well, programmers normally suck at building 3D models even if they can't see this themselves. This holds true for myself at least. Anyway, if you still want to give it a try here is a list of the professional tools used by real 3D artists to build models not only for computer applications but also for special effects for movies:

- Softimage XSI
- Alias Maya
- Discreet 3D Studio Max
- Maxxon Cinema4D

If you want to buy such a program I would recommend you to sell your car first if it is in a good condition. But there are also learning editions available for a cheap price or even for free from those tools which have some restrictions.

Of course there are some low price alternatives too. They are usually enough to play around with a modeling tool a bit to see if you have talent. The following two are 3D model editors available for some 50 bucks and as free shareware versions as well. Both of them are relatively easy to use because they just don't offer that much functionality like a real professional program. But then you won't need all this professional stuff anyway:

- Inivis AC3D (http://www.ac3d.org)
- Chumbalumsoft Milkshape 3D (http://www.milkshape3d.com)

1.1.3 Microsoft's X File Format

The X file format was defined by Microsoft to have a file format for the DirectX SDK which should be heavily supported by Direct3D by providing functions to load such a model for example. I don't want to go into detail

about the complete file format definition but because a lot of file formats are using a similar layout of storing the data I think it is a good idea to show a very simple sample model to you.

Note that you can store a model in a binary file or in an ASCII text file. I use the text file here because then you can just open the model file with a plain text editor and have a look at it. For the Direct3D loader functions the file type does not make a difference though.

The following model is just a sample so that you can have a look at the file format and what it contains. But it represents no valid model data. First, have a look at it and then we will discuss it:

```
xof 0302txt 0064            // header String (has to be first line in
file

Material My_Material_1 {
    1.000000; 0.000000; 0.000000; 1.000000;; // RGBA Diffus
    0.000000;                                // Specular Power
    0.000000; 0.000000; 0.000000;;          // RGB Specular
    0.000000; 0.000000; 0.000000;;           // RGB Emissive
    TextureFilename { „My_Texture1.bmp“; }
    } // Material

Mesh My_Mesh_1 {
     8;                               // number of vertices in mesh
     1.000000; 1.000000;-1.000000;,   // vertex 0
     -1.000000; 1.000000;-1.000000;,   // vertex 1
     -1.000000; 1.000000; 1.000000;,   // vertex 2
      1.000000; 1.000000; 1.000000;,   // vertex 3
      1.000000;-1.000000;-1.000000;,   // vertex 4
     -1.000000;-1.000000;-1.000000;,   // vertex 5
     -1.000000;-1.000000; 1.000000;,   // vertex 6
      1.000000;-1.000000; 1.000000;,   // vertex 7

     12;                               // number of faces in mesh
     3; 0,1,2;,           // face  0: 3 vertices V0, V1 and V2
     3; 0,2,3;,           // face  1: 3 vertices V0, V2 and V3
     3; 0,4,5;,           // face  2: 3 vertices V0, V4 and V5
     3; 0,5,1;,           // face  3: 3 vertices V0, V5 and V1
     3; 1,5,6;,           // face  4: 3 vertices V1, V5 and V6
     3; 1,6,2;,           // face  5: 3 vertices V1, V6 and V2
     3; 2,6,7;,           // face  6: 3 vertices V2, V6 and V7
     3; 2,7,3;,           // face  7: 3 vertices V2, V7 and V3
     3; 3,7,4;,           // face  8: 3 vertices V3, V7 and V4
     3; 3,6,0;,           // face  9: 3 vertices V3, V4 and V0
```

```
    3; 4,7,6;,            // face 10: 3 vertices V4, V7 and V6
    3; 4,6,5;,            // face 11: 3 vertices V4, V6 and V5

   MeshMaterialList {
         2;          // number of used materials
         12;         // number faces with material
         0,          // Face  0 uses material 0
            0,       // Face  1 uses material 0
            0,             // Face  2 uses material 0
            0,             // Face  3 uses material 0
            1,             // Face  4 uses material 1
            1,             // Face  5 uses material 1
            1,             // Face  6 uses material 1
            1,             // Face  7 uses material 1
            1,             // Face  8 uses material 1
            1,             // Face  9 uses material 1
            0,       // Face 10 uses material 0
            0;;            // Face 11 uses material 0

            { My_Material_1 }    // Material 0 (as reference)

            // new material defined without name hence no reference
  possible
            Material {
                    1.000000; 0.000000; 0.000000; 1.000000;;
                    0.000000;
                0.000000; 0.000000; 0.000000;;
                0.000000; 0.000000; 0.000000;;
                TextureFilename { „Meine_Textur3.bmp"; }
                    } // Material
      } // MeshMaterialList

      MeshTextureCoords {
         8;                  // number of texture coords pairs
         0.000000;1.000000;      // Vertex 0 (u;v;) texture coords
         1.000000;1.000000;      // Vertex 1 (u;v;) texture coords
         0.000000;1.000000;      // Vertex 2 (u;v;) texture coords
         1.000000;1.000000;      // Vertex 3 (u;v;) texture coords
         0.000000;0.000000;      // Vertex 4 (u;v;) texture coords
         1.000000;0.000000;      // Vertex 5 (u;v;) texture coords
         0.000000;0.000000;      // Vertex 6 (u;v;) texture coords
         1.000000;0.000000;      // Vertex 7 (u;v;) texture coords
         } // MeshTextureCoords
} // Mesh
```

As you can see there are certain keywords in the format like Material, Mesh, or MeshTextureCoords. Each of the keywords starts the according

data for an object like a material or a mesh and the following data has a certain layout or ordering which must be maintained.

The first top level object in this file is a material definition. After the keyword starting the definition there can also be a name in order to reference to a certain object from inside the file. From the comments you can see what kind of data is included in the object. Note that the commas and semicolons must be set like shown above. After specifying the color values the material object contains a sub-object which is holding the texture file name.

The next top level object is a mesh which must contain at least the coordinates for the vertices as well as the information how to build faces from those vertices. But then this object can contain several sub-objects as well. In this case there are two, namely a list that says which face is using which material and then an object containing the texture coordinates for the vertices. The mesh material list will first say how many materials are used for this mesh, followed by the number of faces which are using a material. Then the materials are listed and the ordering in this list is the index to the face in the face list above. The number in the list says the index to the material that should be used.

Right after this list there is then a list of materials where a material can be defined in one out of two ways. The first material in the list is defined by a reference to the top level object by using its name. The second material is defined right in the list as sub-object. Such a material cannot be used as reference and hence there is no need to give it a name.

The loader functions Direct3D provides for an X file are not smart enough to spot redundant materials using the same texture. If you do not use references to a single material object but specify the same material again and again in the mesh material list then the texture gets loaded redundantly. The render process is also concerned because the mesh is rendered material by material and in the worst case you end up with small batches of triangles rendered in separate calls even if all of them use the same image file for the texture.

And that is all you need to know to understand how an X file is organized. There are a lot of other objects that can be contained in such a file. A very important sub-object for a mesh is for example the list of normal vectors for the vertices. Of course there can be also multiple meshes inside such a file and other things like transformation matrices. But since Direct3D offers us functions to load such a file there is no need to go into more detail beyond what you know now.

1.2 Loading and Displaying Models

Loading and displaying the models will just boil down to open a file that contains a certain level format and then start reading the data from the file. This data is then stored in vertex lists and index lists that you can render using a render device. After all a model is nothing more than just a list of triangles like all geometry. In this paragraph we will discuss first how Direct3D supports you in this process of loading a model. Then we will implement a new render device object that represents a mesh which makes working with models fairly easy.

1.2.1 D3DXMesh Helper Class

Because the X file format for model data was defined especially for the DirectX SDK there is also a class that supports loading and rendering models of this format. Actually, there are a number of classes in the D3DX helper library. The base class for all of those derived classes is the

ID3DXBaseMesh interface. Check the DirectX SDK documentation to see a complete list of functions this interface provides you with. We will use the ID3DXMesh interface here which inherits from the mesh base class because it offers a bit more functionality we will need in this chapter.

Now I don't want to discuss all of the class' functionalities in detail. You do not need much of the functions it provides so I will introduce the functions to you while we are implementing our own class encapsulating this ID3DXMesh interface. There are also several samples in the DirectX SDK that use those mesh classes like *Enhanced Mesh*, *Optimized Mesh*, and *Progressive Mesh*. I recommend you to have a look at those applications as well when you are done reading this chapter.

1.2.1.1 Loading a Model

The following D3DX function lets you load a model in the X file format into an instance of the ID3DXMesh interface:

```
HRESULT WINAPI D3DXLoadMeshFromX(
        LPCTSTR pFilename,      DWORD Options,
        LPDIRECT3DDEVICE9 pDevice, LPD3DXBUFFER *ppAdjacency,
        LPD3DXBUFFER *ppMaterials, LPD3DXBUFFER *ppEffectInstances,
        DWORD *pNumMaterials,    LPD3DXMESH *ppMesh );
```

The parameters of the functions in detail:

- pFileName
  The path of the model file to load.

- Options
  Combination of flags from the enumerated type D3DXMESH. This can be used to specify the mesh as write only, in which pool it should be created, and things like that concerning the internal buffer and texture objects.

- pDevice
  Pointer to a valid Direct3D device that is used for rendering.

- ppAdjacency
  A buffer that is filled with the adjacency information where for each face index three DWORD values will be stored that are indices to neighboring faces.

- ppMaterials
  A buffer that is filled with the model's materials as D3DXMATERIAL structures. This structure contains a D3DMATERIAL9 object and a char string for the image file that is used as texture map on this material.

- ppEffectInstances
  A buffer of effect instances. This is not used here.

- pNumMaterials
  The number of materials in this model file.

- ppMesh
  Reference pointer of the interface type to receive the new object.

As you can see a lot of parameters are using the ID3DXBuffer interface. This interface is just a simple data structure to store any kind of data. You can access the data by calling the following function:

```
void* ID3DXBuffer::GetBufferPointer( void );
```

The returned pointer must then be cast into the type of which the data is so you have to know this format as well as you have to know the number of elements in this buffer. Those buffer objects are used all over the place in the D3DX library.

1.2.1.2 Optimizing a Model

After you have loaded a model and have a valid instance of the ID3DXMesh interface you can then optimize the mesh. Because 3D artists do not care that much about how we render the models they might build the models in a way that is not optimal for rendering. The following function

will control the reordering of the mesh faces and vertices to optimize the performance for rendering:

```
HRESULT ID3DXMesh::OptimizeInplace( DWORD Flags, CONST DWORD
                                    *pAdjacencyIn, DWORD *pAdjacencyOut, DWORD
                                    *pFaceRemap, LPD3DXBUFFER *ppVertexRemap );
```

The parameters of the functions in detail:

- Flags
  One or more flags from the D3DXMESHOPT enumerated type. See below.

- pAdjacencyIn, pAdjacencyOut
  You have to supply the function with the adjacency array you can get from the loading function shown above. If you want to you can also ask the function to return the new adjacency array after optimization or set NULL for this pointer.

  Another way to get the adjacent array for an existing mesh is to call the ID3DXBaseMesh::GenerateAdjacency function. The first parameter is a floating point epsilon value that treats all vertices as coincident whose positions are the same within the epsilon range. The second and last parameter is a DWORD pointer that gets filled with the information.

- pFaceRemap
  This buffer contains a new index for each face located at its old index in the buffer. You can provide NULL here if you don't need that information.

- ppVertexRemap
  This buffer contains a new index for each vertex located at its old index in the buffer. You can provide NULL here if you don't need that information.

The most important thing about this function is of course to name the attributes you want to have optimized. Therefore the following flags are available from the enumerated type D3DXMESHOPT :

- D3DXMESHOPT_COMPACT :
  Reorders faces to remove unused vertices and faces.

- D3DXMESHOPT_ATTRSORT :
  Reorders faces to optimize for fewer attribute bundle state changes and enhanced ID3DXBaseMesh::DrawSubset performance.

- D3DXMESHOPT_VERTEXCACHE :
  Reorders faces to increase the cache hit rate of vertex caches.

- D3DXMESHOPT_STRIPREORDER :
  Reorders faces to maximize length of adjacent triangles.

- D3DXMESHOPT_IGNOREVERTS :
  Optimize the faces only; do not optimize the vertices.

- D3DXMESHOPT_DONOTSPLIT :
  While attribute sorting, do not split vertices that are shared between attribute groups.

- D3DXMESHOPT_DEVICEINDEPENDENT :
  Affects the vertex cache size. Using this flag specifies a default vertex cache size that works well on legacy hardware.

1.2.1.3 Rendering Subsets

After you have successfully loaded and optimized a model as mesh object then you can render it. But as you know by now you can only render triangles in a single call which are using the same texture(s) and the same material. Hence the ID3DXMesh interface does not allow for you to make a single render call. While loading the model you got the number of different materials used where each different texture is also treated with a new material. The following function lets you render the subset of the mesh using a certain material from the mesh:

```
HRESULT ID3DXBaseMesh::DrawSubset( DWORD AttribId );
```

The parameter is just the index of the material ranging from 0, ..., n-1 where n is the number of materials in this mesh. But please note that this call will only render the primitives without activating the material or the texture. In this interface you have to take care of providing the interface and the texture for yourself. The ID3DXMesh interfaces does not do this for you. It will just order the mesh into subsets that can be rendered separately.

1.2.1.4 Attribute Ranges

If you don't want to use the ID3DXBaseMesh::DrawSubset function to render the mesh but want to do this on your own it is still possible. You can get a pointer to the vertex buffer and the index buffer of the mesh by the following functions:

```
HRESULT ID3DXBaseMesh::GetVertexBuffer(LPDIRECT3DVERTEXBUFFER9 *ppVB);
HRESULT ID3DXBaseMesh::GetIndexBuffer(LPDIRECT3DINDEXBUFFER9 *ppIB);
```

But then you would now know how the subsets using different materials and textures are organized in the mesh. So you have to ask this from the mesh as well which is using a table of so called attribute ranges to store this information. Each attribute range object defines the starting location in the vertex buffer and in the index buffer for a single material. The following structure is used in Direct3D for this purpose.

```
typedef struct _D3DXATTRIBUTERANGE {
  DWORD AttribId;        // index of this element in the attribute table
  DWORD FaceStart;       // face start * 3 = start index in index buffer
  DWORD FaceCount;       // number of faces
  DWORD VertexStart;     // start vertex in the vertex buffer
  DWORD VertexCount;     // number of vertices
  } D3DXATTRIBUTERANGE;
```

To get this information from the mesh you can call the following function:

```
HRESULT ID3DXBaseMesh::GetAttributeTable(
            D3DXATTRIBUTERANGE *pAttribTable, DWORD *pAttribTableSize );
```

For the first parameter you have to provide enough memory to take the number of attribute ranges specified in the second parameter. If the first

parameter is NULL then the second parameter is used as output to provide you with the number of materials in the mesh if you don't know this number yet.

1.2.2 Meshes in Sipogen

In the **Sipogen** engine I will implement a render device object to load and display meshes from files using all formats you can write a loader for. Due to our close relationship to DirectX the .x file will naturally be supported by our engine by encapsulating the ID3DXMesh interface. Actually, I will also integrate a lot of functionality from the ID3DXMesh interface into **Sipogen**'s mesh class because internally we will use an instance of this interface to store the data of the 3D model.

1.2.2.1 Render Device Object: Mesh

The **Sipogen** mesh object CMeshRDO is of course implementing an interface called IMesh that is derived from IRenderDeviceObject. I will not show the interface declaration here because you can see the functions it declares by looking at the public virtual functions of the class CMeshRDO implementing the interface. Don't let the complexity of this class scare you away. You can do all kinds of nice things with this class. Loading and rendering 3D models from files is just a minor functionality of this comprehensive mesh object. Among the functionality of this class is intersection detection with bounding boxes versus Rays as well as mesh simplification:

```
class CMeshRDO : public IMesh
{
public:
	CMeshRDO( CRenderDeviceD3D *pDevice );
	virtual ~CMeshRDO();

	virtual bool		Render(bool bOpaqueSubset=true, bool bAlphaSubset=true);

	virtual bool		SetData(	const void *pVerts, VERTEXTYPE Type, unsigned
					int uiNumVerts, const unsigned short *pIndices,
					unsigned int uiNumIndices, const unsigned int
					*pFaceMaterials, const STexturedMat *pMatArray,
```

```
                                    unsigned int uiNumMats );

    virtual bool            Simplify(unsigned int uiDesiredFaces, float fWeightPosition,
                                   float fWeightNormal );

    virtual bool            Intersects( const Vector4 &vcRayOrig, const Vector4
                                     &vcRayDir, float fRayLength, float &fDistance,
                                     const Matrix4x4 *pmWorld=NULL, Vector4
                                     *pvcHit=NULL )const;

    virtual bool            ConvertTo( VERTEXTYPE Type );
    virtual void            CalculateNormals();
    virtual void            Invalidate() { return; }
    virtual void            Restore()    { return; }
    virtual void            GetAabb( Vector4 &vcMax, Vector4 &vcMin, Vector4
                                    *pvcCenter=NULL )const;
    virtual unsigned int    GetNumVertices() const
                            { return m_pMesh->GetNumVertices(); }
    virtual unsigned int    GetNumFaces() const
                            { return m_pMesh->GetNumFaces(); }
    virtual VERTEXTYPE GetVertexType() const { return m_Type; }
    virtual bool            GetData( void *pVerts, unsigned short *pIndices );

    // non-interface methods
    void                    BuildAabb();
    bool                    Load_X( const char *pFile );
    bool                    SetSoftwareProcessing();
    bool                    SetFVF( DWORD dwFVF );
    ID3DXMesh*              GetMeshD3D() { return m_pMesh; }
    unsigned long           GetNumMaterials() { return m_NumMaterials; }
    CTextureRDO*            GetTexture(unsigned long ul) { return m_ppTextures[ul]; }
    CMaterialRDO*           GetMaterial(unsigned long ul) { return m_ppMaterials[ul]; }

    D3DXATTRIBUTERANGE* GetAttributes(unsigned long ul)
                                  { return &m_pAttributes[ul]; }
private:
    CRenderDeviceD3D*           m_pDevice;
    CMaterialRDO**              m_ppMaterials;
    CTextureRDO**               m_ppTextures;
    IDirect3DDevice9*           m_pDeviceD3D;
    ID3DXMesh*                  m_pMesh;
    D3DXATTRIBUTERANGE*         m_pAttributes;
    VERTEXTYPE                  m_Type;
    unsigned long               m_NumMaterials;
    DWORD                       m_dwFVF;
    bool                        m_bSoftware, m_bPrelit;
    Vector4                     m_vcMin, m_vcMax, m_vcCenter;

    bool                        LoadMaterials( D3DXMATERIAL *d3dxMtrls );
    void                        Clear();
```

```
    bool                    CleanupMesh();
};
/*----------------------------------------------------------------------------------------------*/
```

Well, lets discuss this class piece by piece. First, note that it uses the **Sipogen** render device objects for textures and materials of course. The new objects from Direct3D we utilize here are the ID3DXMesh interface as well as the D3DXATTRIBUTERANGE structure. Before we dig into the details have a look at the constructor and the destructor:

```
CMeshRDO::CMeshRDO( CRenderDeviceD3D* pDevice )
{
    m_pDevice           = pDevice;
    m_pDeviceD3D        = pDevice->GetDirect3DDevice();
    m_ppTextures        = NULL;
    m_pMesh             = NULL;
    m_ppMaterials       = NULL;
    m_pAttributes       = NULL;
    m_NumMaterials      = 0;
    m_dwFVF             = D3DFVF_XYZ;
    m_Type              = VTP_XYZ;
    m_bSoftware         = false;
    m_bPrelit           = false;
    m_vcMin.Set( 0.0f, 0.0f, 0.0f );
    m_vcMax.Set( 0.0f, 0.0f, 0.0f );
    m_vcCenter.Set( 0.0f, 0.0f, 0.0f );
}
/*----------------------------------------------------------------------------------------------*/

CMeshRDO::~CMeshRDO()
{
    m_pDevice->NotifyAboutDestruction( this );
    Clear();
}
/*----------------------------------------------------------------------------------------------*/

void CMeshRDO::Clear()
{
    for ( unsigned long i=0; i<m_NumMaterials; i++ )
    {
        if ( m_ppMaterials ) SAFE_DELETE( m_ppMaterials[i] );
        if ( m_ppTextures  ) SAFE_DELETE( m_ppTextures[i]  );
    }

    SAFE_DELETE_A( m_ppTextures );
    SAFE_DELETE_A( m_ppMaterials );
    SAFE_DELETE_A( m_pAttributes );
    SAFE_RELEASE( m_pMesh )
```

```
}
/*----------------------------------------------------------------------------------------*/
```

Again, the constructor and the destructor of this class are boring. The only thing you should note is the notification of the render device when an instance is destroyed to keep the garbage collector informed. The remainder of the destructor is then just taking care of cleaning up.

1.2.2.2 Setting Data for the Mesh Object

Setting the data for the mesh object looks like a complex function call. There are a lot of parameters and some of them seems to be a bit strange. But lets face it piece by piece. The first three parameters deal with the vertices. You have to provide a pointer to an array of vertices, the type of vertex that is used, and finally the number of vertices in the list. The next two parameters are concerned with the indices. You have to hand in a pointer to an array of indices and the number of entries in this array. The final three parameters inform the function about the material and the material distribution. First, there is an array of integer values. The size of the array is the size of the index array divided by 3 which is of course the number of triangles in the mesh. For each triangle you can get the index of the material that should be used from this parameter. The next parameter is an array of STexturedMat instances that contains he actual materials. Finally, the last parameter provides the number of materials in the array.

As you can see this is quite similar to the Direct3D attribute table idea. You are very flexible to set the data for your mesh object once you have the geometry and material information at hand. When we are going to implement model file loaders later in this chapter you will read this information from a file and then call the IMesh::SetData function to fill your mesh object with the information.

*To keep the portability of the **Sipogen** engine as easy as possible our own mesh class will not use Direct3D texture objects or material structures but the engine's interfaces instead.*

The data gets loaded using the ID3DXMesh interface to do the ground work. You already know enough about this interface to understand what is happening here. The model is first loaded as mesh, then it is optimized, and finally its attribute table is stored. But there is another task that needs to be done in this function. You will still remember that the ID3DXMesh interface does not take care of loading or activating material or texture objects for you. It just provides a buffer containing the necessary data but generating those resources is left up to you as well as activated them for rendering of course.

From the comments in the function you can clearly see what is going on inside:

```
bool CMeshRDO::SetData( const void *pVerts, VERTEXTYPE Type, unsigned int
                        uiNumVerts, const unsigned short *pIndices, unsigned int
                        uiNumIndices, const unsigned int *pFaceMaterials,
                        const STexturedMat *pMatArray, unsigned int uiNumMats )
{
    Clear();

    D3DVERTEXELEMENT9 aVertDecl[MAX_FVF_DECL_SIZE];
    m_dwFVF = m_pDevice->MakeDirect3DFormat( Type );
    m_Type  = Type;

    // build vertex declaration from FVF
    if ( FAILED( D3DXDeclaratorFromFVF( m_dwFVF, aVertDecl ) ) ) return false;

    // check for prelit
    if ( m_dwFVF & D3DFVF_DIFFUSE )         m_bPrelit = true;
    else                                    m_bPrelit = false;

    // build the empty mesh object
    if ( FAILED(  D3DXCreateMesh( uiNumIndices/3, uiNumVerts,
            D3DXMESH_MANAGED, aVertDecl, m_pDeviceD3D, &m_pMesh ) ) )
    {
            return false;
    }

    void *_pBuffer = NULL;
```

```
// copy vertex data
if ( SUCCEEDED( m_pMesh->LockVertexBuffer( 0, (void**)&_pBuffer ) ) )
{
        memcpy(_pBuffer,pVerts,m_pDevice->GetVertexSize(Type)*uiNumVerts);
        if ( FAILED( m_pMesh->UnlockVertexBuffer() ) ) return false;
}
else return false;

// copy index data
if ( SUCCEEDED( m_pMesh->LockIndexBuffer( 0, (void**)&_pBuffer ) ) )
{
        memcpy( _pBuffer, pIndices, sizeof( unsigned short ) * uiNumIndices );
        if ( FAILED( m_pMesh->UnlockIndexBuffer() ) ) return false;
}
else return false;

// copy attribute id data (which faces uses which material)
DWORD *_dwBuffer=NULL;

if ( SUCCEEDED( m_pMesh->LockAttributeBuffer( 0, &_dwBuffer ) ) )
{
        for ( unsigned int uiA=0; uiA<(uiNumIndices/3); uiA++ )
        {
                _dwBuffer[uiA] = pFaceMaterials[uiA];
        }
        if ( FAILED( m_pMesh->UnlockAttributeBuffer() ) ) return false;
}
else return false;

D3DXMATERIAL *d3dxMtrls = new D3DXMATERIAL[ uiNumMats ];
m_NumMaterials = uiNumMats;

for ( unsigned int ui=0; ui<uiNumMats; ui++ )
{
    memcpy( &d3dxMtrls[ui].MatD3D.Ambientt, pMatArray[ui].fAmbientRGBA,
                sizeof(D3DCOLORVALUE) );
    memcpy( &d3dxMtrls[ui].MatD3D.Diffuse,  pMatArray[ui].fDiffuseRGBA,
                sizeof(D3DCOLORVALUE) );
    memcpy( &d3dxMtrls[ui].MatD3D.Emissive, pMatArray[ui].fEmissiveRGBA,
                sizeof(D3DCOLORVALUE) );
    memcpy( &d3dxMtrls[ui].MatD3D.Specular, pMatArray[ui].fSpecularRGBA,
                sizeof(D3DCOLORVALUE) );

    d3dxMtrls[ui].MatD3D.Power     = pMatArray[ui].fPower;
    d3dxMtrls[ui].pTextureFilename = pMatArray[ui].sTextureFileName.c_str();
}

// build materials and load textures
if ( !LoadMaterials( d3dxMtrls ) ) { SAFE_DELETE_A( d3dxMtrls ); return false; }
```

```
	SAFE_DELETE_A( d3dxMtrls );

	// optimize the mesh for performance
	ID3DXBuffer *pAdjacency = NULL;
	if (FAILED(D3DXCreateBuffer(uiNumIndices*sizeof(DWORD)*3, &pAdjacency)))
	{
		return false;
	}

	if ( FAILED( m_pMesh->GenerateAdjacency(0.0f, (DWORD*)
					pAdjacency->GetBufferPointer()) ))
	{
		pAdjacency->Release();
		return false;
	}

	if( FAILED( m_pMesh->OptimizeInplace( D3DXMESHOPT_ATTRSORT |
				D3DXMESHOPT_VERTEXCACHE,
				(DWORD*)pAdjacency->GetBufferPointer(), 0, 0, 0 ) ) )
	{
		pAdjacency->Release();
		return false;
	}

	pAdjacency->Release();

	if ( !( m_pAttributes = new D3DXATTRIBUTERANGE[ m_NumMaterials ] ) )
	{
		return false;
	}
	else
	{
		BuildAabb();
		return (m_pMesh->GetAttributeTable(m_pAttributes,&m_NumMaterials )
			== D3D_OK );
	}
}
/*----------------------------------------------------------------------------------------------*/
```

Note that this function is actually not using the interfaces of the **Sipogen** engine for the textures and the materials. Instead it is using the implementing classes directly in order to avoid querying the interfaces from the render device. So the garbage collector does not know about those objects because he was not informed about their creation in the first place. Still, the destructor of the implementing classes will notify the garbage collector about their destruction of course. But the garbage collector would

not care because he has not stored the address of the object and is therefore ignoring the notification and everything is alright – as long as you call the texture and material class destructors manually. This is done in the CMeshRDO::Clear function called from the CMeshRDO destructor.

1.2.2.3 Creating Materials and Textures for the Mesh

Loading the materials and textures is not too complicated. You just have to create a **Sipogen** material RDO fro each Direct3D material that comes as input parameter to the corresponding function. Additionally, you have to try and load the corresponding texture file. Now this is where sometimes problems occur due to the path information. If you fail to load a texture check your paths first with the debugger. But then the CMeshRDO is implemented is a way that allows an application to run even if the texture file is not found. The mesh is then displayed naked – that means without a texture. Here you go:

```
bool CMeshRDO::LoadMaterials( D3DXMATERIAL *d3dxMtrls )
{
      m_ppMaterials = new CMaterialRDO*[ m_NumMaterials ];
      m_ppTextures  = new CTextureRDO*[ m_NumMaterials ];

      unsigned int n = 0;
      bool bAlpha = false;

      // now copy materials and load textures
      for( unsigned long i=0; i<m_NumMaterials; i++ )
      {
            m_ppTextures[i]  = NULL;
            m_ppMaterials[i] = new CMaterialRDO( m_pDevice );

            // note x file material does not contain ambient light so set this manually
            // to values from diffuse light
            memcpy( &(d3dxMtrls[i].MatD3D.Ambient),
                    &(d3dxMtrls[i].MatD3D.Diffuse),
                     sizeof(D3DCOLORVALUE) );

            m_ppMaterials[i]->SetFromD3DMaterial( &d3dxMtrls[i].MatD3D );

            // create texture if present at all
            if ( d3dxMtrls[i].pTextureFilename )
            {
                  m_ppTextures[i] = new CTextureRDO( m_pDevice );
```

```
			if ( m_ppTextures[i] ) // assume TGA textures use alpha channel
			{
				n = (unsigned int)strlen( d3dxMtrls[i].pTextureFilename )-4;
				bAlpha = strcmp(&d3dxMtrls[i].pTextureFilename[n],".tga")==0;

				// else if diffuse material value has alpha
				if ( !bAlpha && (d3dxMtrls[i].MatD3D.Diffuse.a < 1.0f) )
				{
					bAlpha = true;
				}
				if ( !m_ppTextures[i]->SetData(d3dxMtrls[i].pTextureFilename,
								bAlpha ) )
				{
					SAFE_DELETE( m_ppTextures[i] );
				}
			}
			else return false;
		}
	}
	return true;
}
/*----------------------------------------------------------------------------------------------------*/
```

One important thing to point out is that this is the place where the transparency information kicks in. Each subset of a mesh is treated as having transparent parts if either its texture is in .tga format or if its material contains an alpha value other than 1.0f. This is a neat trick of auto-enabling the alpha blending for meshes than use transparency effects.

1.2.2.4 Loading X Files

Due to the comprehensive support of .x files integrated into the D3DX helper library its as easy as it gets to load .x files. The only steps you have to do is to call the D3DXLoadMeshFromX() function, optimize the mesh object, load the materials, and store the attribute table.

```
bool CMeshRDO::Load_X( const char *pFile )
{
	LPD3DXBUFFER pAdjacencyBuffer = NULL, pMtrlBuffer = NULL;
	HRESULT	hr = S_OK;

	Clear();

	if ( FAILED( hr = D3DXLoadMeshFromX( pFile, D3DXMESH_MANAGED,
```

```
                              m_pDeviceD3D, &pAdjacencyBuffer, &pMtrlBuffer, NULL,
                              &m_NumMaterials, &m_pMesh ) ) )
    {
        return false;
    }

    // optimize the mesh for performance
    if( FAILED( hr = m_pMesh->OptimizeInplace( D3DXMESHOPT_COMPACT |
        D3DXMESHOPT_ATTRSORT | D3DXMESHOPT_VERTEXCACHE,
        (DWORD*)pAdjacencyBuffer->GetBufferPointer(), 0, 0, 0 ) ) )
    {
        SAFE_RELEASE( pAdjacencyBuffer );
        SAFE_RELEASE( pMtrlBuffer );
        return false;
    }

    // if there are no materials we are done now
    if ( !pMtrlBuffer || (m_NumMaterials == 0) ) { return true; }

    D3DXMATERIAL* d3dxMtrls = (D3DXMATERIAL*)
                                   pMtrlBuffer->GetBufferPointer();

    if ( !LoadMaterials( d3dxMtrls ) ) { return false; }

    SetFVF( D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1 );
    m_Type = VTP_XYZ_N_1;

    if ( !( m_pAttributes = new D3DXATTRIBUTERANGE[ m_NumMaterials ] ) )
    {
        BuildAabb();
        return (m_pMesh->GetAttributeTable(m_pAttributes,&m_NumMaterials)
            == D3D_OK );
    }
    else return false;
}
/*----------------------------------------------------------------------------------------------*/
```

> *The D3DXATTRIBUTERANGE array stored in the CMeshRDO class is actually not used inside this class at all. But when we are going to work with* tween meshes *later in this book we need to access the attribute range.*

1.2.2.5 Set the FVF for the Mesh

As you can see from the X file model format specification this format is very limited with regard to things like multi-texturing or various vertex

formats – that is unless you introduce and load your own templates from those files. But the ID3DXBaseMesh class offers a function to convert the vertex data of the loaded meshes to a specific format for you. Then you could lock the vertex buffer and fill the appropriate data into the according fields of the vertices.

Here is the function to set a specific FVF for the vertices of the mesh:

```
HRESULT ID3DXBaseMesh::CloneMeshFVF( DWORD Options, DWORD FVF,
                              LPDIRECT3DDEVICE9 pDevice,
                              LPD3DXMESH *ppCloneMesh );
```

The first parameter lets you specify the same flags dealing with the pool and the read and write access like in the D3DXLoadMeshFromX() function. The second parameter is the FVF you want to set for the vertices. The third parameter is the Direct3D device which is used for rendering and the last parameter is a reference pointer to the address where the newly created mesh should be stored.

After you have cloned the mesh to the new FVF format you can of course delete the original object if you don't need this any longer. And here is our function that encapsulates this feature. Note that the function is a public member of the implementing class but it is not part of the interface definition. Hence it is not accessible from outside the **Sipogen** implementation:

```
bool CMeshRDO::SetFVF( DWORD dwFVF )
{
    ID3DXMesh* pTempMesh = NULL;

    if ( m_pMesh )
    {
    if ( FAILED( m_pMesh->CloneMeshFVF( D3DXMESH_MANAGED, dwFVF,
                m_pDeviceD3D, &pTempMesh ) ) )
            return false;
    }

    SAFE_RELEASE( m_pMesh );
    if ( pTempMesh ) m_pMesh = pTempMesh;

    // Compute normals in case the meshes have them
    if ( m_pMesh ) D3DXComputeNormals( m_pMesh, NULL );
```

```
	m_dwFVF = dwFVF;
	return true;
}
/*----------------------------------------------------------------------------------------------*/
```

The final step of this function will calculate the normal vectors for the vertices. This must be done if the former FVF format of the mesh did not have normals but the new FVF of the cloned mesh does have normals for the vertices. The following D3DX function is used to accomplish this task:

```
HRESULT WINAPI D3DXComputeNormals( LPD3DXBASEMESH pMesh,
                                   const DWORD *pAdjacency );
```

The first parameter is the input mesh and the second parameter is the adjacency list of the mesh. If you provide the adjacency list then redundant vertices at the same location but belonging to different faces will be smoothed. Otherwise the normal calculation method is used. Normally, a vertex normal is calculated as averaged normal from the normals of all faces using this vertex.

1.2.2.6 Enforcing Software Vertex Processing

Normally all current graphics adapters support rendering with the fast hardware transformation and hardware lighting where the hardware does the vertex processing. But there are times where you want to switch back to then software vertex processing done by Direct3D as you will see in the next chapter. To enable the mesh to be rendered with software processing you need to add one flag to indicate this usage for the vertex and the index buffer:

```
bool CMeshRDO::SetSoftwareProcessing()
{
	ID3DXMesh* pTempMesh = NULL;

	if ( m_pMesh )
	{
		if ( FAILED( m_pMesh->CloneMeshFVF( D3DXMESH_MANAGED |
		                D3DXMESH_SOFTWAREPROCESSING ,
		                m_dwFVF,m_pDeviceD3D,&pTempMesh)))
		return false;
	}
```

```
    SAFE_RELEASE( m_pMesh );
    if ( pTempMesh ) m_pMesh = pTempMesh;

    if ( m_pMesh ) D3DXComputeNormals( m_pMesh, NULL );

    m_bSoftware = true;
    return true;
}
/*----------------------------------------------------------------------------------------*/
```

This function will clone the mesh again but this time using the additional flag D3DXMESH_SOFTWAREPROCESSING which ensures that the vertex and the index buffer inside the mesh can really be used with software processing. This flag equals the flag D3DUSAGE_SOFTWAREPROCESSING used for creating the vertex buffer and the index buffer that contain the mesh's data.

1.2.2.7 Simplifying the Mesh

One very nice feature and a big argument in the decision to use the ID3DXMesh class internally is the functionality you can get from DirectX. One very useful functionality is the mesh simplification. If you have a mesh with a lot of details and hence a lot of vertices and triangles there are situations where you want to reduce the details and get better performance while rendering the mesh. The D3DXSimplifyMesh function lets you do this. There are a lot of algorithms that let you do mesh simplification on your own. Basically, most of them work by either removing points from the mesh or two points making up an edge. This is called edge removal. Actually, implementing those algorithms is not too difficult. But as always you get trapped in nasty details such as creating the adjacency information, preventing degenerated triangles, and things like that. So using the D3DX functionality saves you a lot of work and headache. I would only recommend implementing your own mesh edge removal function if you are not satisfied with the results you get from the D3DX function.

But before we can start simplifying meshes we need to do a few preparations for the ID3DXMesh object. Don't nail me on that one but the Direct3D documentation suggest that you call a bunch of D3DX functions to make sure the mesh you want to simplify does not contain degenerated

triangles or is bad in another way that would screw up the edge removal algorithm. Here is what you should do first:

- call D3DXCleanMesh()
- remove vertex duplicates by D3DXWeldVertices()
- check mesh integrity using D3DXValidateMesh()

And here is our little helper function that does all those steps for us. For further information on what the functions do in detail you can take a look at the DirectX documentation and get frustrated. The information you can find there is kinda limited.

```
bool CMeshRDO::CleanupMesh()
{
      ID3DXBuffer *pErrors = NULL;
      ID3DXBuffer *pAdj = NULL;
      ID3DXMesh   *pMesh = NULL;

      if ( !m_pMesh ) return false;

      D3DXCreateBuffer( m_pMesh->GetNumFaces()*sizeof(DWORD)*3, &pAdj );
      m_pMesh->GenerateAdjacency( 0.0f, (DWORD*)pAdj->GetBufferPointer() );

      D3DXCleanMesh( D3DXCLEAN_SIMPLIFICATION, m_pMesh,
                              (DWORD*)pAdj->GetBufferPointer(), &pMesh,
                              (DWORD*)pAdj->GetBufferPointer(), &pErrors );

      // remove vertex duplicates
      D3DXWELDEPSILONS wWeights;
      memset( &wWeights, 0, sizeof( D3DXWELDEPSILONS ) );
      wWeights.Position = 1.0f;
      wWeights.Normal   = 1.0f;

      D3DXWeldVertices( pMesh, NULL, &wWeights, (DWORD*)
                              pAdj->GetBufferPointer(), (DWORD*)
                              pAdj->GetBufferPointer(), NULL, NULL );
      // validate the mesh
      if ( FAILED( D3DXValidMesh( pMesh, (DWORD*)pAdj->GetBufferPointer(), 0 ) ))
      {
            return false;
      }

      // store the new, cleaned mesh
      SAFE_RELEASE( m_pMesh );
```

```
    m_pMesh = pMesh;
    m_pMesh->GetAttributeTable( m_pAttributes, &m_NumMaterials );

    return true;
}
/*-------------------------------------------------------------------------------------------*/
```

The next thing you need to know for mesh simplification is the Direct3D structure D3DXATTRIBUTEWEIGHTS. Even if it looks very comprehensive its easy to use. It contains a lot of floating point value fields as you can see from its declaration:

```
typedef struct _D3DXATTRIBUTEWEIGHTS {
    FLOAT Position;
    FLOAT Boundary;
    FLOAT Normal;
    FLOAT Diffuse;
    FLOAT Specular;
    FLOAT Texcoord[8];
    FLOAT Tangent;
    FLOAT Binormal;
} D3DXATTRIBUTEWEIGHTS, *LPD3DXATTRIBUTEWEIGHTS;
```

For each field you can set the weight that is used for the corresponding vertex attribute. If you do edge removal to simplify a mesh each vertex you remove will result in the other vertices to adjust their position and to change their attributes such as position, normal vector, and so on. You can easily calculate an error value that tells you for example by how many percent the changed vertex differs from the original one. The edge removal algorithms will then naturally remove those vertices that introduce the smallest possible error to the mesh. But before evaluating the error values the weights from the D3DXATTRIBUTEWEIGHTS structure are applied to the corresponding error values first. The greater a weight is the greater the corresponding error value gets. If you set a weight to 0.0f then the corresponding attribute is not taken into consideration for the decision. For example, you can set all fields in the structure to 0.0f except for the position and normal field. Set the position value to 0.5f and the normal value to 4.0f. Roughly spoken this would result in the change in position of a vertex to slightly influence the decision whether or not the corresponding edge can be removed while the change in the normal vector will have heavy influence on this decision.

That's quite easy, right? The final piece of information missing is the simplification function itself. And here it is:

```
HRESULT WINAPI D3DXSimplifyMesh(
                        LPD3DXMESH pMesh, const DWORD *pAdjacency,
                        const D3DXATTRIBUTEWEIGHTS *pVertexAttributeWeights,
                        const FLOAT *pVertexWeights, DWORD MinValue,
                        DWORD Options, LPD3DXMESH *ppMesh );
```

The parameters of the functions in detail:

- pMesh, pAdjacency
  The mesh you want to simplify and its adjacency array you can get by calling the ID3DXBaseMesh::GenerateAdjacency function.

- pVertexAttributeWeights
  The attribute weights for each vertex as discussed above. You can also pass in NULL here. This will default the weights with the position, boundary, and normal fields set to 1.0f while the rest of the values is defaulted to 0.0f.

- pVertexWeights
  Additionally, you can provide one weight for each vertex that says how much influence each single vertex should have in the calculation. Normally, you neither want to expose single vertices from your mesh for special treatment nor can you do easily. If you pass NULL the weight 1.0f is used for all vertices.

- MinValue
  The desired minimum value for vertices or triangles – depending on the next parameter. Note that this value can be changed by the function if the desired value does not match the final output of the simplification process.

- Options
  You have to use one out of two options from the D3DXMESHSIMP enumeration concerning the preceding parameter. The two options are D3DXMESHSIMP_VERTEX and D3DXMESHSIMP_FACE.

- ppMesh
  This is the resulting output mesh. Note that the original input mesh stays untouched. After calling this function you have one additional mesh- the simplified one.

Now you know everything you need to know to implement the simplification function. Note that we only expose some of the adjustable parameters for the sake of simplicity in **Sipogen**. The user can only set the number of desired faces after simplification as well as the weights used for the position and the normal attribute:

```
bool CMeshRDO::Simplify( unsigned int uiDesiredFaces, float fWeightPosition,
                         float fWeightNormal )
{
    if ( !CleanupMesh() ) return false;

    ID3DXBuffer *pAdj = NULL;

    // allocate the memory for adjacency buffer
    D3DXCreateBuffer( m_pMesh->GetNumFaces()*sizeof(DWORD)*3, &pAdj );

    // re-generate adjacency information
    if ( FAILED( m_pMesh->GenerateAdjacency( 0.0f, (DWORD*)
                                              pAdj->GetBufferPointer() ) ))
    {
        SAFE_RELEASE( pAdj );
        return false;
    }

    // set simplification parameters
    D3DXATTRIBUTEWEIGHTS attrWeights;
    memset( &attrWeights, 0, sizeof( D3DXATTRIBUTEWEIGHTS ) );
    attrWeights.Position = fWeightPosition;
    attrWeights.Normal  = fWeightNormal;

    // rebuild the mesh object
    D3DVERTEXELEMENT9 aVertDecl[MAX_FVF_DECL_SIZE];
    if ( FAILED( D3DXDeclaratorFromFVF( m_dwFVF, aVertDecl ) ) )
    {
        SAFE_RELEASE( pAdj );
        return false;
    }

    // create a new mesh object which becomes the simplified "copy"
    ID3DXMesh *pMesh = NULL;
    if ( FAILED( D3DXCreateMesh( m_pMesh->GetNumFaces(),
```

```
                                        m_pMesh->GetNumVertices(),
                                        D3DXMESH_MANAGED, aVertDecl,
                                        m_pDeviceD3D, &pMesh ) ))
    {
        SAFE_RELEASE( pAdj );
        return false;
    }

    // finally, simplify the mesh
    HRESULT hr = D3DXSimplifyMesh( m_pMesh, (DWORD*)
                        pAdj->GetBufferPointer(), &attrWeights,
                        NULL, uiDesiredFaces, D3DXMESHSIMP_FACE, &pMesh );

    SAFE_RELEASE( pAdj );

    if ( FAILED( hr ) ) return false;
    else
    {
        SAFE_RELEASE( m_pMesh );
        m_pMesh = pMesh;
        return CleanupMesh();
    }
}
/*----------------------------------------------------------------------------------------*/
```

One situation where you can make good use of this function is the common problem of terrain height map simplification. You can create such a height map with a high resolution loading an 8 bit bitmap or using a noise function for its creation. Next you would build a mesh object for the height map using a very high resolution. Then you can simplify the mesh to reduce the number of faces in the terrain.

> *Besides the simplification mesh the D3DX library also offers the* ID3DXPMesh *to you – a progressive mesh. Additionally to the simplification the* ID3DXPMesh *can add details to a mesh. This is called subdivision because the triangles in the mesh get subdivided. The* ID3DXPMesh *does then store the information about edge removal and subdivision to switch between arbitrary detail levels at runtime. Have a look at this class in the DirectX samples and the documentation.*

1.2.2.8 Testing Intersection with a Ray

Yet another very helpful function is the intersection testing for a mesh. The CMeshRDO can already provide its axis aligned bounding box. You can use this for rough collision detection test. But what you will need in most video game applications is an intersection test between the mesh and a ray. A ray starts at a specific location in space and runs into a specific direction infinitely. The math guys among you ca now start discussing whether or not the ray has an end point in infinity **:-)**

The rest of you will instead listen to situations where an intersection test between such a ray and the mesh is crucial for a video game. One thing that comes to mind is collision detection between an energy weapon beam and a destroyable object for example. Another very common example is the so called ***picking***. One example of picking is when you move your mouse on the screen and click on a 3D object that is then highlighted and marked as selected. This works by casting a ray from the mouse's position into the viewing direction and do an intersection test of this ray with all selectable objects in the scene. Implicit picking is also done if for example the player's cross-hair points onto a selectable object and additional information such as health or name for this object is displayed without the player clicking any button at all.

Again, a lot of functionality you need is provided by the D3DX library. What we do first is to transform the ray into the mesh's coordinate space if necessary. If your mesh is not given in world coordinates but uses a world transformation matrix for correct placement the mesh's coordinate space is not the world space – obviously. But then the ray is given in world space normally. Hence you have to transform both objects into the same coordinate space to do a correct intersection test. And its way better performance to transform the ray into the mesh's coordinate space as opposed to transforming all vertices of the mesh to world space. The **Sipogen** CMeshRDO::Intersects function optionally accepts the mesh's world matrix as parameter. If this is not NULL the matrix is used to transform the ray into the mesh's world space by applying the inverse transformation that moves objects in world space to the mesh's space. Note that the direction of the ray must not undergo translation.

After that there are two functions offered by D3DX to do the actual intersection test.

- D3DXBoxBoundProbe
- D3DXIntersect

The first function checks for an intersection between an axis-aligned bounding box and a ray. The parameters you have to supply are the minimum and the maximum value of the bounding box and the position and direction of the ray. The second function tests for an intersection between an ID3DXBaseMesh object with a ray. As parameters you insert a pointer to the mesh object and the position and direction of the ray. Next is a Boolean parameter that says whether or not there is an intersection. The return value will only tell you if the function call failed. Then there is a list of additional parameters you can use but don't have to such as the closest hit point on the mesh, the list of all hit points – if any – and the distance to the closets hit. We only need to know the distance.

Now I think I did not show you how to calculate an axis-aligned bounding box at all. An axis-aligned bounding box is a box who's edges are parallel to the world coordinate system axis. That means the box's axis are the world coordinate axis as well and that is why you only need to store two points for such a bounding box. The minimum and the maximum point. Each coordinate of those points contains the maximum and the minimum value the box spans on the corresponding axis, respectively. Calculating the bounding box for a mesh is then as simple as you would assume:

```
void CMeshRDO::BuildAabb()
{
	BYTE *pVertices = NULL;
	float *pPosXYZ = NULL;

	m_pMesh->LockVertexBuffer( D3DLOCK_READONLY, (void**)&pVertices );

	// get start values
	pPosXYZ = (float*)pVertices;
	m_vcMax.SetX( pPosXYZ[0] );
	m_vcMax.SetY( pPosXYZ[1] );
	m_vcMax.SetZ( pPosXYZ[2] );
	m_vcMin = m_vcMax;
```

```
    for ( DWORD i=0; i<m_pMesh->GetNumVertices(); i++ )
    {
        // get a float pointer on the vertex data
        pPosXYZ = (float*)pVertices;

        m_vcMax.SetX( max( pPosXYZ[0], m_vcMax.GetX() ) );
        m_vcMin.SetX( min( pPosXYZ[0], m_vcMin.GetX() ) );
        m_vcMax.SetY( max( pPosXYZ[1], m_vcMax.GetY() ) );
        m_vcMin.SetY( min( pPosXYZ[1], m_vcMin.GetY() ) );
        m_vcMax.SetZ( max( pPosXYZ[2], m_vcMax.GetZ() ) );
        m_vcMin.SetZ( min( pPosXYZ[2], m_vcMin.GetZ() ) );

        // set pointer to next vertex
        pVertices += m_pMesh->GetNumBytesPerVertex();
    }

    m_vcCenter = (m_vcMax - m_vcMin) / 2.0f;
    m_pMesh->UnlockVertexBuffer();
}
/*------------------------------------------------------------------------------------------------*/
```

And here is how you implement the intersection test:

```
bool CMeshRDO::Intersects( const Vector4 &_vcRayOrig, const Vector4 &_vcRayDir,
                           float fRayLength, float &fDistance, const Matrix4x4
                           *pmWorld, Vector4 *pvcHit ) const
{
    Vector4 vcRayOrig = _vcRayOrig;
    Vector4 vcRayDir  = _vcRayDir;

    if ( pmWorld )
    {   // transform ray to local space of the mesh
        Matrix4x4 mInv = *pmWorld;
        mInv.InvertTransform();

        vcRayOrig = mInv * vcRayOrig;
        mInv.SetTranslation( 0.0f, 0.0f, 0.0f );
        vcRayDir  = mInv * vcRayDir;
    }

    D3DXVECTOR3 vcxPos(vcRayOrig.GetX(), vcRayOrig.GetY(), vcRayOrig.GetZ());
    D3DXVECTOR3 vcxDir(vcRayDir.GetX(), vcRayDir.GetY(), vcRayDir.GetZ());
    D3DXVECTOR3 vcxMin(m_vcMin.GetX(), m_vcMin.GetY(), m_vcMin.GetZ());
    D3DXVECTOR3 vcxMax(m_vcMax.GetX(), m_vcMax.GetY(), m_vcMax.GetZ());

    // check bounding box for the whole mesh first
    if ( ! D3DXBoxBoundProbe(&vcxMin, &vcxMax, &vcxPos, &vcxDir) ) return false;
```

```
	BOOL bResult = FALSE;
	DWORD dwFace = 0;

	// check all triangles of the mesh for intersection
	if ( FAILED( D3DXIntersect( m_pMesh, &vcxPos, &vcxDir, &bResult, &dwFace,
		NULL, NULL, &fDistance, NULL, NULL ) ) || ( bResult == FALSE) )
	{	// error in function call or no intersection
		return false;
	}
	else
	{	// ray has a limited length
		if ( fRayLength > 0.0f )
		{
			if ( fDistance <= fRayLength )
			{
				if ( pvcHit )
				{	// point of intersection
					(*pvcHit) = vcRayOrig + (vcRayDir * fDistance);
				}
				return true;
			}
			else return false;
		}
		else	// ray has infinite length
		{
			if ( pvcHit )
			{	// point of intersection
				(*pvcHit) = vcRayOrig + (vcRayDir * fDistance);
			}
			return true;
		}
	}
}
/*----------------------------------------------------------------------------------------------*/
```

As you can see I also allow the ray to have a certain length – which is mathematically incorrect but makes sense from a graphical point of view. This way you can easily rule out objects that are too far way from the ray's origin and which you normally don't want pick at all. Mathematically the ray then becomes a line segment. Groovy.

1.2.2.9 Rendering a Mesh Object

Finally, rendering such a mesh object of the **Sipogen** engine is easy. As you know by now the function ID3DXBaseMesh::DrawSubset will render the

geometry of the mesh but with out using a certain texture or material. So the only thing we have to do here is to loop through all materials of the mesh, activate the **Sipogen** material and texture object we created for a certain material of the mesh and then call the function drawing the according subset of the D3DX mesh object.

```
bool CMeshRDO::Render( bool bOpaqueSubset, bool bAlphaSubset )
{
	unsigned int uiRendered = 0;

	if ( !m_pMesh ) return false;

	if ( m_bSoftware && m_pDevice->IsHardware() )
	{
		m_pDeviceD3D->SetSoftwareVertexProcessing( TRUE );
	}

	if ( m_bPrelit )
	{
		m_pDeviceD3D->SetRenderState( D3DRS_LIGHTING, FALSE );
	}
	else
	{
		m_pDeviceD3D->SetRenderState( D3DRS_LIGHTING, TRUE );
	}

	// invalidate current device object settings
	m_pDevice->SetActiveIndexBuffer( NULL );
	m_pDevice->SetActiveVertexBuffer( NULL );

	// draw polygons without alpha values
	if( bOpaqueSubset )
	{
		for( unsigned long i=0; i<m_NumMaterials; i++ )
		{
			// activate material and texture if present and no alpha is used
			if ( (m_ppMaterials[i]&&!m_ppMaterials[i]->HasAlphaComponents()) ||
			     (m_ppMaterials[i] && m_ppTextures[i] && !m_ppTextures[i]
			                                        ->UsesAlpha()) )
			{
				m_ppMaterials[i]->Activate();
				if ( m_ppTextures[i] ) m_ppTextures[i]->Activate( 0 );
				if ( FAILED( m_pMesh->DrawSubset( i ) )) return false;
				uiRendered++;
			}
		}
	}
```

```
	// draw polygons with alpha in material
	if ( bAlphaSubset && (uiRendered < m_NumMaterials) )
	{
		for( unsigned long i=0; i<m_NumMaterials; i++ )
		{
		   // activate material and texture if present and alpha is used
		   if ( (m_ppMaterials[i] && m_ppMaterials[i]->HasAlphaComponents()) ||
		       (m_ppMaterials[i] && m_ppTextures[i] && m_ppTextures[i]
		                                                  ->UsesAlpha()) )
		     {
		         m_ppMaterials[i]->Activate();
		         if ( m_ppTextures[i] )  m_ppTextures[i]->Activate( 0 );
		         if ( FAILED( m_pMesh->DrawSubset( i ) ))  return false;
		     }
		}
	}

	if ( m_bSoftware && m_pDevice->IsHardware() )
	{
		m_pDeviceD3D->SetSoftwareVertexProcessing( FALSE );
	}

	if ( m_bPrelit ) m_pDeviceD3D->SetRenderState( D3DRS_LIGHTING, TRUE );
	return true;
}
/*------------------------------------------------------------------------------------------------*/
```

This function is separating the drawing of the subsets using an alpha value other than 1.0 in the material or where the texture uses values from its alpha channels. This way you can render the solid part or the transparent part of the mesh only if you want to. This might be helpful to avoid certain visual artifacts. Normally, you have to render all solid objects in a scene before rendering the transparent ones.

1.3 Sipogen's Mesh Factory

Now you have seen how **Sipogen** enables you to use its mesh object to store the geometrical data and texture information of a 3D model. Also, the IMesh interface supports ***Microsoft's*** .x file format. But of course that is not enough. As of now the following steps are required to load a model from an .x file:

```
spg::IMesh *pMesh = (spg::IMesh*)pSipogenDevice->CreateObject(spg::RDO_MESH);
```

```
if ( pMesh )
{
	if ( pMesh->Load_X( "tiger.x" ) )
	{
		return true;
	}
}
```

That's quite complex already because we can do better than that. The user of the engine should have as few lines of code as possible. If you want to load a model why create a mesh instance first. There must be a part of the engine that can do this job for you. Also, if your model is not in the .x format you are lost. You would have to use the IMesh::SetData call to manually feed the data you have read from your file to the mesh interface. This will work of course but there are a number of common formats which the engine should support. What I want the engine to let the user do is this:

```
spg::IMesh *pMesh = NULL;

if ( m_pSipogenDevice->GetMeshFactory()->CreateMesh( "tiger.x", &pMesh ) )
{
	return true;
}
```

This looks much cleaner to me than the version presented beforehand. And that is what a so called mesh factory is used for in **Sipogen**. The mesh factory has to recognize which format the user wants to load by looking at the file ending. Then it has to parse the file and feed the data to a new instance of the spg::IMesh interface. The more file formats you implement into the mesh factory the better **Sipogen**'s functionality will be. First, I want to show you the mesh factory implementation, after that I will discuss one model file loader out of the three formats **Sipogen** supports in this book's version. Those three format are:

- Microsoft .x
- Autodesk 3D Studio Max .3ds
- Inivis AC3D .ac

It's a good guess that we are now adding an IMeshFactory interface to the Sipogen engine. This interface does only offer two functions. One to load static mesh models and the other one to load tweening models. But you can find out more about that in the succeeding paragraph. Lets concentrate on the normal static models for now. Here is the interface declaration:

```
class IMeshFactory
{
      public:
       IMeshFactory() {};
       virtual ~IMeshFactory() {};

      virtual bool CreateMesh( const std::string &sFileName, IMesh **ppOut ) const=0;

      virtual bool CreateTweenMesh( const std::string &sFileName,
                                    ITweenMesh **ppOut ) const=0;
};
/*----------------------------------------------------------------------------------------*/
```

The class implementing this interface ain't much more complicated either. It just adds one protected function and one protected member to the class:

```
class CMeshFactory : public IMeshFactory
{
public:
      CMeshFactory( CRenderDeviceD3D *pDevice ) { m_pDevice = pDevice; }
      virtual ~CMeshFactory() { ; }

      virtual bool     CreateMesh( const std::string &sFileName, IMesh **ppOut )const;

      virtual bool     CreateTweenMesh( const std::string &sFileName,
                                        ITweenMesh **ppOut )const;
protected:
       std::string     GetFileEnding( const std::string &sFileName )const;

       CRenderDeviceD3D*  m_pDevice;
};
/*----------------------------------------------------------------------------------------*/
```

Please note that the CRenderDeviceD3D class now gets a new member m_pMeshFactory as an instance of this class. The only thing is does with this new member is to initialize and destroy it on initialization and destruction, respectively. Finally, it also allows access to this member via the new derived IRenderDevice::GetMeshFactory function.

Now lets look at the protected function of this class. Its purpose is to extract the file ending of the given input string. It does this by searching of for the last dot in the string and returning the characters that appear after that dot:

```
std::string CMeshFactory::GetFileEnding( const std::string &sFileName ) const
{
	size_t i = sFileName.find( std::string(".") );

	if ( i != std::string::npos )
	{
		std::string sResult = sFileName.substr( i );
		std::transform( sResult.begin(), sResult.end(), sResult.begin(), std::tolower );
		return sResult;
	}
	else
	{
		return std::string("");
	}
}
/*---------------------------------------------------------------*/
```

Quite simple but very helpful as you can see in the public function used to create a static model. Look at it first and we shall then discuss it afterwards:

```
bool CMeshFactory::CreateMesh( const std::string &sFileName, IMesh **ppOut ) const
{
	std::string sFileType = GetFileEnding( sFileName );

	if ( !sFileType.empty() && m_pDevice )
	{
		CMeshRDO *pMesh = new CMeshRDO( m_pDevice );
		(*ppOut) = pMesh;

		if ( !pMesh ) return false;

		// FILE FORMAT: .x ( Microsoft DirectX )
		if ( sFileType == std::string(".x") )
		{
			if ( !pMesh->Load_X( sFileName.c_str() ) )
			{
				SAFE_DELETE( pMesh );
			}
		}
		// FILE FORMAT: AC3D ( Inivis Ltd. AC3D )
		else if ( (sFileType == std::string(".ac")) || (sFileType == std::string(".ac3d")) )
```

```
		{
			CModelLoaderAC3D ac3dLoader( m_pDevice );
			if ( !ac3dLoader.LoadAllObjects( sFileName, ppOut ) )
			{
				SAFE_DELETE( pMesh );
			}
		}
		// FILE FORMAT: 3DS ( 3D Studio Max )
		else if ( (sFileType == std::string(".3ds")) || (sFileType == std::string(".3DS")) )
		{
			CModelLoader3DS maxLoader( m_pDevice );
			if ( !maxLoader.LoadAllObjects( sFileName, ppOut ) )
			{
				SAFE_DELETE( pMesh );
			}
		}

		if ( pMesh )
		{	// put into list for garbage collecting
			m_pDevice->NotifyAboutCreation( pMesh );
			return true;
		}
		else return false;
	}
	else return false;
}
/*------------------------------------------------------------------------------------------*/
```

Here you can see how the different file formats get loaded with the mesh factory. The .x format is internally supported so there is no need to do anything else than just calling the spg::CMeshRDO::Load_X function. For all other formats you need to supply a file loader class. The next paragraph introduces you to one comprehensive example how to write such a file loader. After reading this paragraph you should have no problems adding your own favorite file format to the engine without changing the engine's interfaces at all.

1.4 Sipogen's Model Loader

With the IMesh interface at hand the **Sipogen** engine is well equipped to let the user efficiently store and access 3D geometry. But hard-coding this geometry or creating it algorithmically such as spheres is not a good way to go for most detail objects. Normally, you employ a 3D artist who creates

detailed 3D models for you. This artist then saves his work in a specific file format. Your task is now to parse this file and extract the geometry information from it.

For this purpose **Sipogen** contains a model loader base class. This class does nothing else than providing a few helper functions especially for parsing ASCII files. Lets look at this base class first before we start implementing a real file loader as derived class.

1.4.1 CModelLoader Base Class

The most important piece of information each loader in the **Sipogen** engine needs to know is the render device. From this render device pointer the loader instance can build an IMesh instance for example. Other that this attribute the loader base class offers just a few static functions you can use to parse ASCII files.

```
class CModelLoader
{
public:
	CModelLoader() { m_pDevice = NULL; }
	virtual ~CModelLoader() { ; }

	// ASCII HELPER FUNCTIONS
	static bool OpenFile( const std::string &name, std::ifstream &ifs,
	                      bool bTrashWhiteSpaces=true );
	static bool SkipPossibleComment( std::string &sToken, std::ifstream &ifs );
	static bool GetNextToken( std::string &sToken, std::ifstream &ifs );
	static bool GetTokenAsFloat( float &fToken, std::ifstream &ifs );
	static bool GetTokenAsInt( int &iToken, std::ifstream &ifs );
	static bool SkipNextCharIfItIs( const char &chAwaited, std::ifstream &ifs );

protected:
	CRenderDeviceD3D*   m_pDevice;
};
/*----------------------------------------------------------------------------------------------*/
```

I won't go into detail about those functions. They are written to work with STL input file stream objects. The names of the functions are very obvious and tell you what the functions are supposed to do. You can look at the implementation in the accompanying source code files.

1.4.2 Loading Inivis AC3D Files

The first level format I want to introduce here besides .x is the *Inivis AC3D* format. AC3D stands for Andy Colebourne who is the programmer of the tool. Its flexible to some degree and you can easily create static 3D models with a steep learning curve. Writing plug-ins such as custom file exporters is also possible.

> *You can check out the pretty nice modeling package AC3D on its website http://www.ac3d.org.*

The file format is very straight forward which is good because it is easy to understand but which is bad because it requires rearranging the data a lot. First, the file format is ASCII which means its just plain text you can open with any text editor of your choice. Basically, the .ac format just looks like this:

```
AC3Db
MATERIAL
[…]
OBJECT
[…]
```

The first line is the header which contains the keyword AC3D followed by a hex-number stating the version. For example, 0xb is version 1.1. After the header line there are an arbitrary number of materials. After the materials, there is an arbitrary number of objects. Those objects then contain the actual geometry. *AC3D* does also support a hierarchy of objects. But first see how a MATERIAL data set looks like:

```
MATERIAL %s %f %f %f  %f %f %f  %f %f %f  %f %f %f  %d %f
```

Right after the keyword MATERIAL there is a string that is the material's name. Then there are four triples of floating point values that define the material light properties in the following order:

- diffuse color
- ambient color

- emissive color
- specular color

The last two values are an integer value that defines the shininess of the material and a floating point value that defines the material's transparency. After reading all materials from the file you will find the geometry objects. Their definition looks like this:

```
OBJECT %s
    *name %s
    *data %d
    *texture %s
    *texrep %f %f
    *rot %f %f %f  %f %f %f  %f %f %f
    *loc %f %f %f
    *url %s
    *numvert %d
        numvert lines of %f %f %f
    *numsurf %d
        *SURF %d
        *mat %d
        refs %d
            refs lines of %d %f %f
    kids %d
```

The keyword OBJECT is followed by the name string of the object. This name could be world which is the root object's name. This root object is always present in an *AC3D* file even if there is no other geometry. We don't have to care about that one. The root object does only contain the kids member which says how many objects follow as children of the current object. For the root object the number of children is the number of top level hierarchy objects in the whole file.

The data keyword is followed by an integer number saying how many characters of a so called data string follow in the next line. This string is pretty cool because in *AC3D* you can write an arbitrary string for each object. That allows you to add certain attributes to your objects which *AC3D* does not support as such but that you need for your own video game. For example you could specify the objects number of hit points or whether it should cast a shadow or not.

Next in line is the name string of the texture used for the object and the two floating point values used for the texture repeat on the object. The rot and loc keywords specify the rotation and location of the object you need to transform its coordinates with. The url string is yet another string. Finally, numvert tells you the number of vertices of the objects followed by all vertices' coordinates. After that, numsurf tells you the number of faces in this object followed by the definition of a face for each face in the object. This definition first contains an array of flags, then the index of the material used for this surface, and finally the number of references and a list of references. A reference is what we call an index, so the reference is an index to the object's vertex list as well as a pair of texture coordinates.

And that's it. The *AC3D* file format is quite easy, right? So lets start parsing it. Here is the class we define for loading *AC3D* files.

```
class CModelLoaderAC3D : public CModelLoader
{
public:
	CModelLoaderAC3D( CRenderDeviceD3D *pDevice );
	virtual ~CModelLoaderAC3D();
	virtual bool LoadAllObjects( const std::string &sFileName, IMesh **ppTarget );

private:
	class Face
	{
	public:
		Face() { m_vIndis.reserve(12); m_iNum=0; m_matID=-1; }
		virtual ~Face() { ; }
		inline void	AddIndex( int i ) { m_vIndis.push_back( i ); m_iNum++; }
		inline void	SetMaterialID( int id )	{ m_matID = id; }
		inline void	SetTexturedMaterialID( int id ) { m_TexMatID = id; }
		inline void	SetTextureID( const std::string s )	{ m_sTexture = s; }
		inline int	GetMaterialID() const	{ return m_matID; }
		inline std::string GetTextureID() const	{ return m_sTexture; }
		inline int	GetNumIndices() const	{ return m_iNum; }
		inline int	GetNumTris() const	{ return m_iNum-2; }
		inline int	GetTexturedMaterialID() const { return m_TexMatID; }
		inline int	GetIndex( int i ) const	{ return m_vIndis[i]; }
	private:
		std::vector<int>	m_vIndis;
		std::string	m_sTexture;
		int	m_iNum, m_matID, m_TexMatID;
	};

	class Material
```

```
{
public:  /* [...] only accessor functions */
private:
        float    m_fDiffuse[3], m_fEmissive[3], m_fAmbient[3], m_fSpecular[3];
        float    m_fShine, m_fAlpha;
};

int                                  m_iNumVertices, m_CurrentOffset;
int                                  m_iNumTris;
std::vector<Face>                    m_vFaces;
std::vector<Material>                m_vMaterials;
std::vector<IMesh::STexturedMat>     m_vTexMats;
std::vector<SVertex_XYZ_N_1>         m_vVerts;
unsigned int*                        m_pFaceMatIndis;

bool          CreateTexturedMaterialList();
bool          CreateOutputMesh( IMesh **ppTarget );
bool          LoadMaterial( std::ifstream &ifs );
bool          LoadObject( std::ifstream &ifs, Matrix4x4 *pmParent );
bool          LoadObjectFace( std::ifstream &ifs, Face &face, float fOffU,
                              float fOffV, float fRepU, float fRepV );

inline void   AddVertex( const SVertex_XYZ_N_1 &v )
                  { m_vVerts.push_back(v); ++m_iNumVertices; }
inline void   AddFace( const Face &f )
                  { m_vFaces.push_back(f); m_iNumTris += f.GetNumTris(); }
};
/*----------------------------------------------------------------------------------------------*/
```

The class already looks quite complex. It contains two embedded classes used to store faces and materials, respectively. Note that for the sake of space requirements I have cut off the functions from the embedded Material class. There are only accessor functions that let you set and get the member attributes ad you will see those functions in action when we are reading the file in a minute.

1.4.2.1 Extracting Materials and Objects

The CModelLoaderAC3D::LoadAllObjects function is the main parser function that loops through the whole file. It skips the first line in the file which is the header of an *AC3D* file. After that it loops through all lines of the files and searches for the keywords MATERIAL and OBJECT. Upon finding one of them it calls the corresponding workhorse function that loads the material's or the object's data from the file. After loading all objects and

all materials from the file we still need to rearrange the data. First, we need to create a list of textured materials the IMesh interfaces wants to have. Second, we have to assemble the output mesh. At this point we have only all the single faces as separate objects and need to create an IMesh instance from those faces. Here is the function:

```
bool CModelLoaderAC3D::LoadAllObjects( const std::string &sFileName, IMesh **ppM )
{
    std::ifstream ifs;
    if ( !OpenFile( sFileName, ifs ) ) return false;
    else
    {
        std::string sToken;

        // first ist header "AC3Dx" with hex number for version
        if ( !GetNextToken( sToken, ifs ) ) return false;

        while ( GetNextToken( sToken, ifs ) )
        {
            if ( sToken == std::string( "MATERIAL" ) )
            {
                if ( !LoadMaterial( ifs ) ) return false;
            }
            else if ( sToken == std::string( "OBJECT" ) )
            {
                if ( !LoadObject( ifs, NULL ) ) return false;
            }
        }

        // all data collected, now build mesh
        CreateTexturedMaterialList();

        return CreateOutputMesh( ppM );
    }
}
/*----------------------------------------------------------------------------------------------------*/
```

Reading a material instance or an object instance from the file is similar straightforward. You can expect a certain keyword at a specific location in the file. Upon finding the keyword read in the following data and store it in your own data structures. The CModelLoaderAC3D::LoadObject function is quite a workhorse in this implementation because it reads most of the data belonging to the OBJECT keyword. It won't print this function here as well as I won't waste pages for the function reading a MATERIAL instance. I assume that you are advanced enough to see how you can parse and

extract the file data here. Or you are just smart enough to look at the source code accompanying this book and see how it works. Basically, it just reads the data, builds a transformation matrix from the object's location and rotation, and then reads all vertices. Each vertex is transformed using the transformation matrix and then added to the global list of vertices. For each object the number of vertices read from the file before parsing the current object is stored as m_CurrentOffset. This allows you to adjust the face indices not to be relative to the face vertex list any longer but relative to the global vertex list of the whole model instead.

1.4.2.2 Reading Face Data

The only part of an OBJECT instance that is not wholly handled in the workhorse function CModelLoaderAC3D::LoadObject is the data of a face. If you have read the number of faces in the object then you run a loop for this count and call the following function that extracts the face's data.

The input parameters to this function are the stream and a new, empty instance of the embedded Face class as well as the texture repeat and offset used for the face. This is where it gets a bit messy in *AC3D*. The texture repeat and offset are stored for each OBJECT but the real texture coordinates are not stored with the vertices but for each index. The indices in turn are stored in the face information. That is why you need to feed the repeat and offset data to this function.

```
bool CModelLoaderAC3D::LoadObjectFace( std::ifstream &ifs, Face &face, float fOffU,
                                        float fOffV, float fRepU, float fRepV  )
{
    std::string sToken;
    int iInt=0, iTex=0;
    int iMaterial=-1;
    float fU=0.0f, fV=0.0f;

    // KEYWORD: SURF
    if ( !GetNextToken( sToken, ifs ) || sToken != std::string("SURF") ) return false;
    else if ( !GetNextToken( sToken, ifs ) ) return false;

    // KEYWORD: mat
    if ( !GetNextToken( sToken, ifs ) || sToken != std::string("mat") ) return false;
    else
    {
```

```
		if ( !GetTokenAsInt( iMaterial, ifs ) ) return false;
		else face.SetMaterialID( iMaterial );
	}

	// KEYWORD: refs (num texture coordinates)
	if ( !GetNextToken( sToken, ifs ) || sToken != std::string("refs") ) return false;
	else
	{
		if ( !GetTokenAsInt( iTex, ifs ) ) return false;

		for ( int i=0; i<iTex; i++ )
		{
			if ( !GetTokenAsInt( iInt, ifs ) ) return false;
			if ( !GetTokenAsFloat( fU, ifs ) ) return false;
			if ( !GetTokenAsFloat( fV, ifs ) ) return false;

			face.AddIndex( iInt + m_CurrentOffset );

			m_vVerts[ m_CurrentOffset + iInt ].fUV0[0] = fOffU + fRepU*fU;
			m_vVerts[ m_CurrentOffset + iInt ].fUV0[1] = -(fOffV + fRepV*fV);
		}
	}
	return true;
}
/*----------------------------------------------------------------------------------------*/
```

Each index read from the file gets incremented by the current offset. The current offset is the number of vertices that are stored in the class' global vertex list before you read the current object's vertices. The reason for this is that the indices in the file are relative to the vertex list of the current object. If you have the index 0 for example it refers to the first vertex of the current object. But those vertices are stored in the global vertex list which already contains for example 200 vertices of preceding objects from the same file. So you need to store the object relative index 1 as index 201 to reference the correct vertex in the global vertex list.

The second catch you need to take into consideration is that *AC3D* uses OpenGL and its right handed coordinate system. To correct that you need to multiply the z coordinate of all vertices and the V texture coordinate with –1 to let the object appear as it should in the left handed coordinates system Direct3D uses.

1.4.2.3 Creating the Material List

After parsing the whole .ac file you have stored all materials in Material instances of the embedded class. But what you need o have is an array of IMesh::STexturedMat instances to feed the IMesh::SetData function. Furthermore you need to provide a list which faces are using which material from this array. The CModelLoaderAC3D::CreateTextureMaterialList function loops through all faces read from the file. For each face it loops through all IMesh::STexturedMat instances you have created so far (which is none to start with). If the function finds the correct material for the face it will store the corresponding index for the face. If a correct instance cannot be found the function creates a new IMesh::STexturedMat instance and stores the new index for the Face instance.

```
bool CModelLoaderAC3D::CreateTexturedMaterialList()
{
    IMesh::STexturedMat TexMat;
    std::vector<int> viMatID;
    float fRGBA[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
    int iNum=0, iMatID=0, iFaces = (int)m_vFaces.size();
    bool bFound=false;

    for ( int i=0; i<iFaces; ++i )
    {
        bFound = false;
        iMatID = m_vFaces[i].GetMaterialID();

        for ( int j=0; j<iNum; ++j )
        {
            if ( (viMatID[j] == iMatID ) && (m_vTexMats[j].sTextureFileName
                == m_vFaces[i].GetTextureID()) )
            {
                m_vFaces[i].SetTexturedMaterialID( j );
                bFound = true;
                break;
            }
        }
        if ( !bFound )
        {
            m_vMaterials[iMatID].GetAmbient( fRGBA );
            memcpy( TexMat.fAmbientRGBA, fRGBA, sizeof(float)*4 );
            m_vMaterials[iMatID].GetDiffuse( fRGBA );
            memcpy( TexMat.fDiffuseRGBA, fRGBA, sizeof(float)*4 );
            m_vMaterials[iMatID].GetEmissive( fRGBA );
            memcpy( TexMat.fEmissiveRGBA, fRGBA, sizeof(float)*4 );
            m_vMaterials[iMatID].GetSpecular( fRGBA );
            memcpy( TexMat.fSpecularRGBA, fRGBA, sizeof(float)*4 );
            TexMat.fDiffuseRGBA[4] = m_vMaterials[iMatID].GetAlpha();
```

```
                TexMat.fPower = m_vMaterials[iMatID].GetShine();
                TexMat.sTextureFileName = m_vFaces[i].GetTextureID();

                m_vTexMats.push_back( TexMat );
                m_vFaces[i].SetTexturedMaterialID( iNum );
                viMatID.push_back( iMatID );
                ++iNum;
            }
        }
        return true;
}
/*----------------------------------------------------------------------------------------------*/
```

One problem with the *AC3D* format is that the material itself does not store a texture. That means one material could be used multiple times with different textures. But that is not what the IMesh interface wants to see. The IMesh interface connects each material to a certain texture. That means if you have multiple faces in the .ac file using the same material but different textures you have to create one unique IMesh::STexturedMat instance for each. That is why the function shown above compares the material ID as well as the texture name.

1.4.2.4 Creating the Output Mesh

Once you have read all data from the file and assembled the material list you can create the output mesh. Before you can hand over the data to the IMesh::SetData function there are two more things left to do. First, you have to create an integer array that says for each face in the face list which material ID applies for that face. The second thing you need to do here depends on the fact that the face data given in the *AC3D* file defines the indices for the corresponding face. But *AC3D* allows for n-sided faces which is quite bad for us. In Direct3D you can only render 3-sided faces which are also know as triangles. To create the triangulation for an n-sided face is not that complicated if you are not dealing with non-convex faces. Then you can simply build a triangle fan for the face where the first vertex of he face is the first vertex of each triangle in the triangulation. Each succeeding pairs of indices are then the additional two indices for each triangle in the triangulation.

And here is the implementation:

```
bool CModelLoaderAC3D::CreateOutputMesh( IMesh **ppTarget )
{
	if ( *ppTarget )
	{
		bool bResult = false;
		unsigned int uiNumI = m_iNumTris*3;
		unsigned int uiNumMaterials = (int)m_vTexMats.size();
		unsigned short *pIndices = new unsigned short[ uiNumI ];

		// build face material indices for triangulation
		m_pFaceMatIndis = new unsigned int[m_iNumTris];
		for ( int f=0, cf=0; f<(int)m_vFaces.size(); f++ )
		{
			for ( int t=0; t<m_vFaces[f].GetNumTris(); t++, cf++ )
			{
				m_pFaceMatIndis[cf] = m_vFaces[f].GetTexturedMaterialID();
			}
		}

		// sort out index data as triangle list
		int iFaces = (int)m_vFaces.size();
		int iI = 0;
		unsigned short i0=0, i1=0, i2=0;
		for ( int i=0; i<iFaces; i++ )
		{
			for ( int t=0; t<m_vFaces[i].GetNumTris(); t++ )
			{	// revert ordering to convert left handed
				pIndices[iI] = m_vFaces[i].GetIndex( 2+t ); ++iI;
				pIndices[iI] = m_vFaces[i].GetIndex( 1+t ); ++iI;
				pIndices[iI] = m_vFaces[i].GetIndex(  0  ); ++iI;
			}
		}

		bResult = (*ppTarget)->SetData( &m_vVerts[0], VTP_XYZ_N_1,
						(unsigned int)m_iNumVertices,
						pIndices, uiNumI, m_pFaceMatIndis,
						&m_vTexMats[0], uiNumMaterials );
		SAFE_DELETE_A( pIndices );
		return bResult;
	}
	else return false;
}
/*------------------------------------------------------------------------------------------*/
```

Finally, the IMesh::SetData function is called and all the information you read from the *AC3D* file is presented in the correct form for this function to create the corresponding **Sipogen** mesh object for you.

A word of warning about the current .ac loader which is not stable for all AC3D files. AC3D allows you to build non-convex faces that screw up our triangulation. Be sure to load models with only convex faces or adjust the triangulation algorithm (ear clipping e.g.). Another unsupported feature is the grouping option in AC3D that lets you group objects together introducing OBJECT instances called "group". Before loading your AC3D file select all objects and ungroup all … or handle the group objects in your loader code.

1.4.3 Loading Discreet 3D Studio Max Files

The second 3D model file format I want to discuss here is .3ds. As opposed to the *AC3D* file format it's a binary file format. Due to space consideration I cannot discuss the loader's source code in detail here. But you can look at the full working code on the accompanying source code project. As of now the loader only supports static models. *Discreet*'s .3ds format can also contain key-frame information for animated models but read more about that at the end of paragraph 1.5.

The reason for choosing the .3ds format here is because it is very popular and a lot of tools can export to this file format. That means you will fid a number of free models throughout the internet to test your loader and do nice demos for example.

Basically, the .3ds file format is very simple. It is chunk-based that means all information inside the file is packed into so called chunks. A chunk in a .3ds file looks like this in pseudo-code:

```
Chunk
{
    ChunkHeader
    {
        ID;          // 2 bytes integer
        Length;      // 4 bytes integer, value incl. header, data, and sub-chunks
    };

    Data;
    SubChunks [];
```

```
};
```

A chunk is nothing else than a data container. The big advantage of chunk based formats is that each chunk can tell you its overall size. That means if you find a chunk inside the file which you don't know or don't want to read you can just skip it because you know where it ends in the file. That is why each chunk comes with a header. The header contains the chunk ID which tells you what kind of data is stored right after the header. The second information you can get from the header is the length of the whole chunk. Note that this length includes the 6 bytes of the header as well as the size of all sub-chunks. And that's true, a chunk can contain as many sub-chunks as you want it to.

1.4.3.1 Main Chunks in .3ds Files

The idea of a chunk-based format is not that hard to follow I think. So lets have a look at the most important chunks in the .3ds format. Note that there are a hell of a lot of different chunks for this format and I possibly can't cover all of them here. Neither do you need to know all of them. But for more comprehensive descriptions of the file format refer to the very good document *3D-Studio File Format* rewritten by Martin van Velsen and the very good reference *The Unofficial 3DStudio 3DS File Format* written by Jeff Lewis. Especially the latter one lists nearly all types of chunks you can find in a .3ds file. You can find those documents all over the internet by using your favorite search engine.

Anyway, the most important chunks you will need to read are the following ones. This list is taken in slightly changed form the spacesimulator.net website that provides a small kick off tutorial to get you started with loading the .3ds format. Before you start to explore the source code companion to this book you should maybe look at this online tutorial to get a basic idea of a .3ds file loader. But first look at the most important chunks:

```
MAIN CHUNK                         0x4D4D
  3D EDITOR CHUNK                  0x3D3D
    OBJECT BLOCK                   0x4000
      TRIANGULAR MESH              0x4100
        VERTICES LIST              0x4110
        FACES DESCRIPTION          0x4120
```

FACES MATERIAL	0x4130
MAPPING COORDINATES LIST	0x4140
SMOOTHING GROUP LIST	0x4150
LOCAL COORDINATES SYSTEM	0x4160
MATERIAL BLOCK	0xAFFF
MATERIAL NAME	0xA000
AMBIENT COLOR	0xA010
DIFFUSE COLOR	0xA020
SPECULAR COLOR	0xA030
TEXTURE MAP 1	0xA200
BUMP MAP	0xA230
REFLECTION MAP	0xA220
[SUB CHUNKS FOR EACH MAP]	
MAPPING FILENAME	0xA300
MAPPING PARAMETERS	0xA351
KEYFRAMER CHUNK	0xB000
MESH INFORMATION BLOCK	0xB002
FRAMES (START AND END)	0xB008
OBJECT NAME	0xB010
OBJECT PIVOT POINT	0xB013
HIERARCHY POSITION	0xB030

From most of the chunk descriptions you should already see what they are used for. All geometry of the model is stored in a TRIANGULAR_MESH chunk. The materials are stored in MATERIAL_BLOCK chunks.

> *The names used for the chunks vary with each document you read about this file format. Don't worry about them too much, a chunk is identified by the hexadecimal ID value, not by its name.*

The information in the KEYFRAMER section of the file might suggest that you only need to read this if you want to read files with key-frame animation. But note that this is not totally true. The pivot-point information is crucial even for static models. From the LOCAL_COORDINATES chunk you can get a transformation matrix that is already applied to the vertices stored in the corresponding TRIANGULAR_MESH object. But if the 3D artist used different pivot points for the meshes you need to inverse the vertex transformations, subtract the pivot point and reapply the transformation to the vertices. That is why you still need to read the key-frame information for static models.

1.4.3.2 Deriving the .3ds Loader Class

Before we can start loading chunks I want to show you the class that will do this task and that is derived from the CModelLoader base class. As you can see it's a bit more complex than the one that loads *AC3D* files. There are more subroutines to load various chunks as well as there are more embedded data structures. The reason is of course that the format is more complex than the .ac format. This also requires you to store the data in a more or less straight forward manner and reorganize the data after loading to be able to create fuel an IMesh instance with the data. Here is the class:

```
class CModelLoader3DS : public CModelLoader
{
public:
    CModelLoader3DS( CRenderDeviceD3D *pDevice );
    virtual ~CModelLoader3DS();

    virtual bool LoadAllObjects( const std::string &sFileName, IMesh **ppTarget );
    virtual bool LoadKeyframed( const std::string &sFileName, ITweenMesh **Target );

    struct Object
    {
        Object() { pParent=0; Angle=0.0f; StartVertex=0; EndVertex=0; }
        std::string             Name;
        unsigned int            StartVertex, EndVertex;
        Matrix4x4               Matrix, mParentCombined;
        spg::Vector4            Pivot, PosTrack, ScaleTrack, Axis;
        float                   Angle;
        Object*                 pParent;
        std::vector<Object*>    vChilds;
    };

    struct ChunkHeader { unsigned int ChunkID; unsigned long ChunkSize; };
    struct TagHeader { short iFlags, aUnknown[4], iNumKeys, iUnknown; };
    struct VectorEntry { short iFrameNumber; long  lUnknown; float fXYZ[3]; };
    struct AngleAxisEntry { short iFrameNumber; long lUkwn; float fAngle, fXYZ[3]; };
    struct Hierarchy  { short ID; int   ObjectID; };

private:
    int                                 m_iNumVertices, m_CurrentOffset;
    std::vector<Object>                 m_vObjects;
    std::vector<Hierarchy>              m_vHierachy;
    std::vector<IMesh::STexturedMat>    m_vTexMats;
    std::vector<std::string>            m_sMatNames;
    std::vector<SVertex_XYZ_N_1>        m_vVerts;
    std::vector<unsigned short>         m_iIndices;
```

```
    std::vector<unsigned int>           m_iFaceMatIndex;
    unsigned int*                       m_pFaceMatIndis;
    float                               m_fScaling;

    unsigned char  ReadChar( std::ifstream &file );
    unsigned int   ReadInt( std::ifstream &file );
    unsigned long  ReadLong( std::ifstream &file );
    void           ReadChunkHeader( std::ifstream &file, ChunkHeader *h );
    std::string    ReadName( std::ifstream &file, unsigned long &lSizeRead );
    bool           ReadMaterial( std::ifstream &file, unsigned long lSizeToRead );
    bool           ReadTriMesh( std::ifstream &file, unsigned long lSizeToRead );
    void           ReadKFObjInfo( std::ifstream &file, unsigned long lSizeToRead );
    bool           ReadColor( std::ifstream &file, unsigned long lSizeToRead,
                              unsigned long &lSizeRead, unsigned char *RGB );
    void           AddFace( int i0, int i1, int i2, bool bAddOffset=true );
    int            GetTexturedMaterial( const std::string &sMaterial );
    int            GetObjectIndex( const std::string &sObject );
    void           IntegrateToHierarchy( Hierarchy h );
    bool           CreateOutputMesh( IMesh **ppTarget );
    inline void    AddVertex( const SVertex_XYZ_N_1 &v )
                   { m_vVerts.push_back(v); ++m_iNumVertices; }
};
/*------------------------------------------------------------------------------------------*/
```

The most important data structure here is the Object structure. It contains the information for a single TRIANGULAR_MESH chunk such as the start vertex in the global vertex list of the CModelLoader3DS class as well as the end vertex. The other important information is the hierarchy information and the pivot point. As mentioned above you need to change the vertex transformation if there is a pivot point other than (0,0,0). As far as the hierarchy is concerned don't worry about that too much for static models.

A triangular mesh instance in a .3ds file stores its vertices in transformed world coordinates. It also contains the transformation matrix that was used to transform the vertices. In the keyframer chunk you can find position, scaling, axis, and angle information. From this information you can also build a transformation matrix for the mesh relative to its parent mesh. Calculating the combined transform down from the root to the mesh you get the same matrix that is stored in the mesh object.

1.4.3.3 Extracting the Main-Chunks

I won't go into much detail about loading the .3ds file. This would require some dozens of pages to print all the source code here. After all it just boils down to rearranging the data you get from the chunks. If you look at the source code companion of this book you won't have any problem seeing what's going on in the code. So I will just show you the main function that is used to read the file from the top level point of view, calling subroutines to load specific chunks. Once you see how it works you know all of the subroutines quite well because they all look very similar except for the data that you get from a chunk.

```
bool CModelLoader3DS::LoadAllObjects( const std::string &sFileName, IMesh **Target )
{
	std::ifstream file( sFileName.c_str(), std::ios::binary );
	ChunkHeader header={ 0, 0 };

	while ( file )
	{
		ReadChunkHeader( file, &header );

		switch( header.ChunkID )
		{
			case MAIN_CHUNK:		// no data to read
			case EDITOR_CHUNK:		// no data to read
			case KEYFRAMER_CHUNK:		// no data to read
			break;

			case MESH_INFORMATION_BLOCK:
			{
				ReadKFObjInfo( file, header.ChunkSize - 6 );
				break;
```

```
				}
				case VERSION:			// we support version >= 3
				{
					unsigned long v = ReadLong( file );
					if ( v < 3 ) return false;
					break;
				}
				case SCALING_FACTOR:	// scaling for the whole file
				{
					file.read( (char*)&m_fScaling, 4 );
					break;
				}
				case OBJECT_BLOCK:
				{
					unsigned long iDummy=0;
					Object obj;
					obj.Name = ReadName( file, iDummy );
					obj.StartVertex = (unsigned int)m_vVerts.size();
					obj.EndVertex = obj.StartVertex;
					obj.vChilds.reserve( 5 );
					m_vObjects.push_back( obj );
					break;
				}
				case MATERIAL_BLOCK:
				{
					if ( !ReadMaterial( file, header.ChunkSize - 6 ) ) return false;
					break;
				}
				case TRIANGULAR_MESH:
				{
					if ( !ReadTriMesh( file, header.ChunkSize - 6 ) ) return false;
					break;
				}
				default:
				{
					file.seekg( header.ChunkSize - 6, std::ios::cur );
					break;
				}
			}
		}
		return CreateOutputMesh( ppTarget );
}
/*----------------------------------------------------------------------------------------------------*/
```

1.4.3.4 Creating the Output Mesh

Assuming that you have now read and reorganized all the data from the 3D model you can now create the output mesh. But there is one last step

we already discussed. You need to cancel out the pivot point. In the following function that fills the IMesh instance with the data you can see how it first applies the inverted transformation of an object to its vertices. Then the pivot point gets subtracted and the transformation is applied again. After that you are ready to build your **Sipogen** mesh.

```
bool CModelLoader3DS::CreateOutputMesh( IMesh **ppTarget )
{
    bool bResult = false;
    SVertex_XYZ_N_1 v;
    Vector4 vc;
    Matrix4x4 mInv;

    // take care of pivot points
    for ( std::vector<Object>::iterator itObj = m_vObjects.begin();
          itObj != m_vObjects.end(); itObj++ )
    {
        if ( ! itObj->Pivot.IsNull() )
        {
            mInv.InvertTransform( itObj->Matrix );
        }

        for ( unsigned int i=itObj->StartVertex; i<=itObj->EndVertex; i++ )
        {
            if ( ! itObj->Pivot.IsNull() )
            {
                v = m_vVerts[ i ];
                vc.Set( v.fX, v.fY, v.fZ );

                vc = vc * mInv;
                vc = vc - itObj->Pivot;
                vc = vc * itObj->Matrix;

                v.fX = vc.GetX();   // invert Y/Z due to left handed coord sys
                v.fY = vc.GetZ();
                v.fZ = vc.GetY();
                m_vVerts[ i ] = v;
            }
            else    // just switch Y and Z due to left handed coord sys
            {
                float fZ = m_vVerts[ i ].fZ;
                m_vVerts[ i ].fZ = m_vVerts[ i ].fY;
                m_vVerts[ i ].fY = fZ;
            }
        }
    }

    bResult = (*ppTarget)->SetData( &m_vVerts[0], VTP_XYZ_N_1,
```

```
                                        (unsigned int)m_iNumVertices,
                                        &m_iIndices[0],
                                        (unsigned int)m_iIndices.size(),
                                        &m_iFaceMatIndex[0],
                                        &m_vTexMats[0],
                                        (unsigned int)m_vTexMats.size() );
    return bResult;
}
/*----------------------------------------------------------------------------------------------------*/
```

Okay, I know that was quite a run through this format. But with all the information you can find here and the resources I pointed out to you there is no problem in understanding the whole source code of the .3ds loader. The comments throughout the source code should also give you answers if you get stuck somewhere. Note that this loader does only support static models. Also, I found some problems with some models in rare occasions which I could not track down so far. But for most free models you can find all over the internet this loader should work fine.

And now its time to leave he yard of the static models and visit their neighbors – the key-frame animated 3D models. To display such a key-framed animation you need to know about the so called ***vertex tweening*** which is similar to morphing.

1.5 Vertex Tweening in Direct3D

Vertex tweening describes a method to do a linear blend between two sets of vertices containing different data but where the number of vertices is identical. This can be used for example to build a model in two different poses and then *tween* between those two poses. Maybe you now the dolphin sample from the DirectX SDK which shows a dolphin swimming in the water with his tail fin moving. Hint: If you don't know the sample then start it now to see what I mean.

Actually, this animation of the dolphin is done using vertex tweening with three different models showing the dolphin in three different poses. The technique works by linear interpolation between the vertex attributes like the position and the normal vectors. So you have to supply a blending factor that is a weighting for the influence of one of the two poses where

the other pose is weighted with the inverse factor of course. Now it is of course possible that you load two models showing two different poses of the same model as meshes, get their vertex buffer content and do the interpolation for each frame on your own. Then fill the interpolated data into a new vertex buffer and render the model.

But there is no need to do this because either your graphics adapter can do this or Direct3D could handle the tweening for you. In fact both solutions are not very different when it gets down to the source code. At first you have to evaluate whether your graphics adapter supports this calculation. Lets suppose a class has an attribute m_pDeviceD3D which is a valid Direct3D device and a bool variable m_bHardwareTweening which should receive the result of the evaluation:

```
D3DCAPS9 caps;
m_pDeviceD3D->GetDeviceCaps( &caps );

if ( caps.VertexProcessingCaps & D3DVTXPCAPS_TWEENING )
  {
  m_bHardwareTweening = true;
  }
else m_bHardwareTweening = false;
```

As you can see the device caps structure can be checked with the tweening cap of Direct3D to see if the graphics adapter can do vertex tweening in hardware. If the hardware is not able to do this then Direct3D can take over if you just let it do so. There are no more changes necessary than just switching from hardware vertex processing to software vertex processing. As you remember our engine initializes Direct3D with mixed vertex processing so you can just toggle from hardware vertex processing to software vertex processing at run time like so:

```
if ( !m_bHardwareTweening )
  m_pDeviceD3D->SetSoftwareVertexProcessing( TRUE );
```

Now you can just proceed with vertex tweening in the way you would do if the hardware has had support for vertex tweening. But now Direct3D will calculate this for you instead. If you want to switch back to hardware vertex processing you would call the same function but set FALSE as parameter.

To adjust the vertex tweening you would then need to set two render-states no matter if the hardware will do the vertex tweening or if Direct3D took things over. Supposed you have a float value fTween between 0.0f and 1.0f which should be used to weight the first pose with fTween and the second pose with 1-fTween then you can activate the tweening by setting the following two render-states:

```
m_pDeviceD3D->SetRenderState( D3DRS_TWEENFACTOR, *((DWORD*)&fTween) );
m_pDeviceD3D->SetRenderState( D3DRS_VERTEXBLEND, D3DVBF_TWEENING );
```

The render-state D3DRS_TWEENFACTOR is used to let you set the factor that is used as weight in the blending operation between the two meshes. Note that you will end up with one of the original poses if the factor is 0 or 1. The second render-state D3DRS_VERTEXBLEND is used to activate the vertex blending and you can set a value from the enumerated type D3DVERTEXBLENDFLAGS:

```
typedef enum _D3DVERTEXBLENDFLAGS {
  D3DVBF_DISABLE = 0,
  D3DVBF_1WEIGHTS = 1,
  D3DVBF_2WEIGHTS = 2,
  D3DVBF_3WEIGHTS = 3,
  D3DVBF_TWEENING = 255,
  D3DVBF_0WEIGHTS = 256
  } D3DVERTEXBLENDFLAGS;
```

The value D3DVBF_DISABLE would result in no blending operation. Only the world transformation matrix will be used for the vertex processing. Now it is also possible to set multiple world matrices between which a blending if performed based on weights saved in the vertex structure. this is what the values 1, 2, and 3 are used for. The value D3DVBF_TWEENING will blend two sets of geometry using the tweening factor set in the render-state D3DRS_TWEENFACTOR as shown above. The value D3DVBF_0WEIGHTS is used to apply a single blending matrix with a weight of 1.0f. For the remainder of this book we will stick to vertex tweening only. The following line of codes deactivates the vertex blending:

```
m_pDeviceD3D->SetRenderState( D3DRS_VERTEXBLEND, D3DVBF_DISABLE );
```

The vertex tweening is a nice feature for really simple animations as opposed to the vertex blending which would require each vertex to have a

weight as attribute. Note that you can do more advanced forms of blending by using vertex shaders. But we won't cover this here.

There is one thing left to do if you want to have vertex tweening. You still need to tell Direct3D where the two different sets of geometry come from which Direct3D should blend together for you. The next paragraph will be the key to this secret.

1.6 Using multiple Streams in Direct3D

Using DIrect3D a graphics adapter offers multiple streams to the programmer which can be fed with data. Such a stream is a pipeline from the application to the graphics adapter and it is used to send vertex data and all kind of vertex attributes for processing the geometry. Just take a look at the function you are using to activate a vertex buffer:

```
HRESULT IDirect3DDevice9::SetStreamSource( UINT StreamNumber,
                    IDirect3DVertexBuffer9 *pStreamData,
                    UINT OffsetInBytes, UINT Stride );
```

The first parameter of the function is the number of the stream you want to use to send the vertex buffer's data down to the graphics adapter. Normally you will use 0 here because the flexible vertex format (FVF) does not let you define the usage of multiple streams for the vertex data.

The number of available streams on a graphics adapter can be found by looking at the field MaxStreams of the D3DCAPS9 structure. Normally for DirectX 9 compatible graphics adapters this should be in the range 1-16 so you can have up to 16 vertex buffers active at the same time used in a single render call.

1.6.1 Advantages of multiple Streams

The advantages of using multiple streams are twofold. The first interesting usage for multiple vertex buffers is that you can store the lists of vertex

attributes separately and not as a list of vertex structures. So instead of having just one array of the type VERTEX you can have an array of the type Vector3 to hold the position data, one array of the type Vector3 to hold the normal vectors, one array of floating point values to hold the texture coordinates and so on.

The advantage of such an approach is that you can separate the attributes from a vertex structure. If you look at most model file formats for example then you can see that those files store the data in a similar way. Not grouped into vertices but into lists of positions, normal vectors, texture coordinates, and so on. This is also more easily to convert to OpenGL by the way as opposed to a vertex structure containing all attributes for a single vertex. So by using multiple streams you can get rid of the predefined vertex types and the FVF definitions by using vertex declarations.

But before we come to that I want to name the second big advantage of using multiple streams. You can now load two different poses of the same model into one vertex buffer each. Then you activate both vertex buffers for two different streams at the same time. This enables you to do vertex tweening between those two poses which is effectively a so called key-frame animation that is interpolating between two key poses of a model. The Figure 1 shows two key-frame positions of a bird. The top most pose is the key-frame with the wings in their upper positions while the lowest pose is the key-frame with the wings in their lowest position. Those are the only two poses of the model you have to build and to set for two different streams and the same time. All other poses in between are calculated by vertex tweening and create a smooth animation sequence.

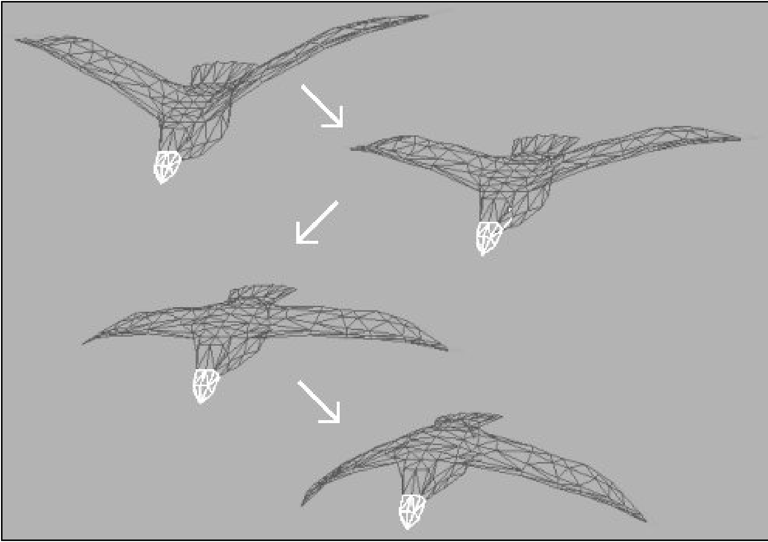

Figure 1: Interpolating between two key-frame positions

1.6.2 Vertex Declarations instead of FVF

Until now we only used the flexible vertex format definitions (FVF) to build our custom vertex types. But there is one thing you cannot do with the FVF and that is defining the source of the data for the graphics adapter. So all the information for a vertex has to sit in the stream with the index 0 which is the default stream for the data. The second way to define the layout of a vertex is to use a vertex declaration instead of the FVF. A vertex declaration is an object of the interface called IDirect3DVertexDeclaration9 and can be created with this function:

```
HRESULT IDirect3DDevice9::CreateVertexDeclaration(
                    CONST D3DVERTEXELEMENT9* pVertexElements,
                    IDirect3DVertexDeclaration9** ppDecl );
```

In the second parameter you supply the address of a reference pointer where the new vertex declaration object should be stored. For the first parameter you hand over an array of D3DVERTEXELEMENT9 structures which describes the requested vertex layout. Before I go into detail about this structure I want to show you how to activate a vertex declaration:

```
HRESULT IDirect3DDevice9::SetVertexDeclaration(
                    IDirect3DVertexDeclaration9* pDecl );
```

Note that a succeeding call to IDirect3DDevice::SetFVF will invalidate the active vertex declaration the same way as this call invalidates the active FVF because both are meant for the same purpose and only one vertex layout description can be active at a given time.

So here is the structure that describes the vertex layout when you are defining a vertex declaration. You have to define one instance of this structure for each data element that is used for the vertex data to be processed, hence the need for an array of this structure:

```
typedef struct _D3DVERTEXELEMENT9 {
  WORD Stream;
  WORD Offset;
  BYTE Type;
  BYTE Method;
  BYTE Usage;
  BYTE UsageIndex;
  } D3DVERTEXELEMENT9;
```

The structure's fields in detail:

- Stream
  The index of the stream where this vertex attribute will be located.

- Offset
  The offset in bytes from the beginning of the stream to this data element.

- Type
  The data type of this element from the enumerated type D3DDECLTYPE. The most important ones are:
  D3DDECLTYPE_FLOAT3: Array of three 32 bit floating points values.
  D3DDECLTYPE_FLOAT4: Array of four 32 bit floating points values.
  D3DDECLTYPE_D3DCOLOR: A 32 bit color value like a D3DCOLOR or DWORD.

- Method
  This is only used for tessellation of geometry. Because we don't use this here just set the method to D3DDECLMETHOD_DEFAULT.

- Usage
  Defines the usage of the element to tell the graphics adapter what kind of information it is. One of the enumerated type D3DDECLUSAGE (see below).

- UsageIndex
  This is the index of the usage type which means that you can have multiple elements defining the same usage but with different indices.

The last field in this declaration is the key to the reason why we have to use a vertex declaration for vertex tweening with two sets of geometry. With a FVF you cannot do something like this:

```
DWORD dwFVF = ( D3DFVF_XYZ | D3DFVF_XYZ | D3DFVF_DIFFUSE );
```

But you would need to do this in order to specify that you have two positions to do the interpolation. When using a vertex declaration you can in fact define two vertex elements that have the usage type for the position but that have different usage indices like 0 and 1 for example. Then the vertex tweening can be done.

So here are the types you can define for the usage:

```
typedef enum _D3DDECLUSAGE {
    D3DDECLUSAGE_POSITION = 0,          // position data
    D3DDECLUSAGE_BLENDWEIGHT = 1,       // weight in blending
    D3DDECLUSAGE_BLENDINDICES = 2,      // index for vertex blending
    D3DDECLUSAGE_NORMAL = 3,            // normal vector
    D3DDECLUSAGE_PSIZE = 4,             // point sprite size
    D3DDECLUSAGE_TEXCOORD = 5,          // pair of texture coordinates
    D3DDECLUSAGE_TANGENT = 6,           // tangent vector (bump mapping)
    D3DDECLUSAGE_BINORMAL = 7,          // binormal vector (bump mapping)
    D3DDECLUSAGE_TESSFACTOR = 8,        // for geometry tessellation
    D3DDECLUSAGE_POSITIONT = 9,         // transformed position
    D3DDECLUSAGE_COLOR = 10,            // diffuse color
    D3DDECLUSAGE_FOG = 11,              // fog value
    D3DDECLUSAGE_DEPTH = 12,            // depth value
    D3DDECLUSAGE_SAMPLE = 13            // sample for displacement mapping
```

```
} D3DDECLUSAGE;
```

The following code snippet shows you how an FVF for a rather simple vertex type would translate to an array of vertex elements used to create a vertex declaration:

```
DWORD dwFVF = ( D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1 );

D3DVERTEXELEMENT9 sVertexElement [] = {
  { 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_POSITION, 0 },
  { 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_NORMAL,0},
  { 0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
     D3DDECLUSAGE_TEXCOORD, 0 },
  { 0xFF, 0, D3DDECLTYPE_UNUSED, 0, 0, 0 }
  };
```

As you can see you can put different vertex attributes into different streams. If you define a vertex element array in the following way than you can use separate vertex buffers for the position data, for the normal data, and for the texture coordinates. This little code snippet shows you how to do this:

```
IDirect3DVertexDeclaration9* m_pVertexDecl = NULL;

D3DVERTEXELEMENT9 sVertexElement [] = {
  { 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_POSITION, 0 },
  { 1, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_NORMAL, 0 },
  { 2, 0, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
    D3DDECLUSAGE_TEXCOORD, 0 },
  { 0xFF, 0, D3DDECLTYPE_UNUSED, 0, 0, 0 }
  };

m_pd3dDevice->CreateVertexDeclaration( sVertexElement, &m_pVertexDecl );
m_pd3dDevice->SetVertexDeclaration( m_pVertexDecl );

m_pd3dDevice->SetStreamSource( 0, m_pVB_Pos, 0, sizeof(float)*3 );
m_pd3dDevice->SetStreamSource( 1, m_pVB_Normals, 0, sizeof(float)*3 );
m_pd3dDevice->SetStreamSource( 2, m_pVB_TexCoords, 0, sizeof(float)*3 );
```

Now you can render as you did before when using a FVF for the vertex layout description. But the graphics adapter will now take the data from

different streams and not from a single stream with a big array of vertex structures.

> *Each array of D3DVERTEXELEMENT9 structures must be using a closing element as shown in the code snipped above. Because of that you don't have to provide the number of elements in the array. Note that Direct3D defines the macro D3DDECL_END() for exactly this line which you can use as well.*

Now you know everything there is to know to get started with implementing vertex tweening in the **Sipogen** engine. Note that this tweening should only be used for fairly simple animations.

1.7 Tweened Meshes in Sipogen

You have just seen how to create a mesh object in which you can load either custom geometry data or a model in the .x format from a file. In this paragraph we will add a render device object to the **Sipogen** engine that lets you load an arbitrary number of data set or model files into one object – you can think of this object as array of meshes. But there is one restriction to the models you add to this array: The number of vertices and faces of all meshes in the array must not be different from each other. Ideally, all meshes in the array show exactly the same 3D model but in different poses. And now you should easily be able to guess where we are heading right now.

Once all models in the array are in place you can render one representation of the model by selecting two meshes from the array and specifying a blending factor that is used for vertex tweening. The vertex tweening is the same what you might know as morphing. That means for each vertex in the model you have two different positions (from the two meshes) and do an interpolation between both positions using the blend factor. The result of this tweening operation is a representation of the model that is a mixture of both input meshes. Or to put it in more clear words: The result is an intermediate pose of the model showing the model

in the transition from one mesh pose to another one. The blend factor in the range of [0,1] tells you about the progress of the transition.

So the tween mesh we are about to implement now can be used for simple animated models also know as key-frame animation.

1.7.1 Render Device Object: TweenMesh

The **Sipogen** mesh object TweenMesh which can use vertex tweening is of course implementing an interface called ITweenMesh that is derived from IRenderDeviceObject. Again, I won't show the interface definition in this book but the implementing class is just using public functions declared in the interface. There are no other public non-interface or private functions needed for this class.

```
class CTweenMeshRDO : public ITweenMesh
{
public:
      CTweenMeshRDO( CRenderDeviceD3D *pDevice );
      virtual ~CTweenMeshRDO();

      virtual bool    Render( float fTween, bool bOpaqueSubset=true, bool
                              bAlphaSubset=true );

      virtual bool    RenderKeyframe( unsigned int uiNum, bool bOpaqueSubset=true,
                                      bool bAlphaSubset=true );

      virtual bool    SetData( const std::vector<std::string> &vModelNames );
      virtual void    Invalidate() { return; }
      virtual void    Restore()    { return; }

      virtual bool    Intersects( float fTween, const Vector4 &vcRayOrig, const Vector4
                                  &vcRayDir, float fRayLength, float &fDistance, const
                                  Matrix4x4 *pmWorld=0, Vector4 *pvcHit=0 )const;

      virtual bool    Intersects( unsigned int uiKeyFrame, const Vector4 &vcRayOrig,
                                  const Vector4 &vcRayDir, float fRayLength, float
                                  &fDistance, const Matrix4x4 *pmWorld=0,
                                  Vector4 *pvcHit=0 )const;

      virtual void    GetAabb( float fTween, Vector4 &vcMax, Vector4 &vcMin,
                               Vector4 *pvcCenter=NULL )const;

      virtual void    GetAabb( unsigned int uiKeyFrame, Vector4 &vcMax, Vector4
```

```
                                        &vcMin, Vector4 *pvcCenter=NULL )const;
private:
     CRenderDeviceD3D*              m_pDevice;
     IDirect3DDevice9*              m_pDeviceD3D;
     std::vector<CMeshRDO*>         m_vMeshs;
     unsigned long                  m_ulNumVertices;
     unsigned long                  m_ulNumTriangles;
     unsigned int                   m_uiFPS;
     bool                           m_bHardwareTweening;
     IDirect3DVertexDeclaration9 *m_pVertexDeclaration;
};
/*----------------------------------------------------------------*/
```

To keep the necessary work down to a small level this class is using the CMeshRDO class to load the two model files between which this class should do the vertex tweening. Note the vertex declaration belonging to this class.

The constructor of this class will now not only initialize the attributes to initial values but it will also check whether the graphics adapter supports vertex tweening in hardware or if we need to switch to software vertex processing when rendering the tweened mesh. The vertex declaration is also created here and now you can see how to use the usage index field to have separate data coming from to streams but which are meant for the same usage.

```
CTweenMeshRDO::CTweenMeshRDO( CRenderDeviceD3D* pDevice )
  {
  m_pDevice         = pDevice;
  m_pDeviceD3D      = pDevice->GetDirect3DDevice();
  m_ulNumVertices   = 0;
  m_ulNumTriangles  = 0;

  m_pVertexDeclaration = NULL;

  D3DCAPS9 caps;
  m_pDeviceD3D->GetDeviceCaps( &caps );
  if ( caps.VertexProcessingCaps & D3DVTXPCAPS_TWEENING )
       m_bHardwareTweening = true;
  else m_bHardwareTweening = false;

  D3DVERTEXELEMENT9 decl[] =
    {
    // First stream is first mesh
    { 0,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_POSITION, 0},
```

```
    { 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_NORMAL,   0},
    { 0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_TEXCOORD, 0},
    // Second stream is second mesh
    { 1,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_POSITION, 1},
    { 1, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_NORMAL,   1},
    { 1, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_TEXCOORD, 1},
    D3DDECL_END()
    };

  m_pDeviceD3D->CreateVertexDeclaration( decl, &m_pVertexDeclaration );
  }
/*---------------------------------------------------------------------------------------------*/

// destructor
CTweenMeshRDO::~CTweenMeshRDO_X()
{
      m_pDevice->NotifyAboutDestruction( this );
      SAFE_RELEASE( m_pVertexDeclaration );
}
/*---------------------------------------------------------------------------------------------*/
```

The destructor is then just telling the garbage collector about its death and releases all allocated memory.

1.7.2 Set Data for the TweenMesh Object

To set the data for a tweened mesh you have to supply an array of name strings. Each string in the array is the file name of the corresponding model you want to load as mesh. What the function does is to loop through the array and try to create one CMeshRDO instance for each entry in the array using **Sipogen**'s mesh factory. After loading one model the function will then compare the vertex count and the face count of the model with those of the first model in the array. This is to ensure that both models have the same number of vertices and indices. Otherwise a tweening would not be possible or won't make sense at least resulting in undefined behavior – which is usually bad of course.

After loading the key-frames as separate IMesh *objects you have to check whether or not hardware tweening is possible on the given hardware. If not do not forget to call the function* IMesh::SetSoftwareProcessing*. Otherwise Direct3D might fail rendering the meshes.*

And believe it or not, that is all you have to do to load all the data necessary for key-framed animation. Thanks to our comprehensive mesh class we save a lot of work here for loading the different poses:

```
bool CTweenMeshRDO::SetData( const std::vector<std::string> &vModelNames )
{
      m_vMeshs.clear();
      std::string sName;

      for ( unsigned int ui=0; ui<(unsigned int)vModelNames.size(); ui++ )
      {
            IMesh* pMesh = NULL;

            if ( !m_pDevice->GetMeshFactory()->CreateMesh( vModelNames.at( ui ),
                                                            &pMesh ) )
            {
                  return false;
            }
            else
            {
                  CMeshRDO* pMeshRDO = (CMeshRDO*)pMesh;

                  // set software processing if no hardware tweening available
                  if (!m_bHardwareTweening&&!pMeshRDO->SetSoftwareProcessing())
                  {
                        return false;
                  }

                  // get counters from first key-frame mesh
                  if ( m_vMeshs.size() == 0 )
                  {
                        m_ulNumVertices  = pMeshRDO->GetMeshD3D()->
                                                            GetNumVertices();
                        m_ulNumTriangles = pMeshRDO->GetMeshD3D()->
                                                            GetNumFaces();
                  }
                  // check if all keyframes use the same counts
                  else if (
            (m_ulNumVertices  != pMeshRDO->GetMeshD3D()->GetNumVertices()) ||
            (m_ulNumTriangles != pMeshRDO->GetMeshD3D()->GetNumFaces()) )
```

```
		{
			return false;
		}

		m_vMeshs.push_back( pMeshRDO );
	}
}
return true;
}
/*-----------------------------------------------------------------------------------------------------------------------*/
```

One very important thing here is to check whether or not the vertex tweening can be done in hardware. If this is not the case you need to adjust the two meshes used to hold the two key-frames between which we want to implement to software vertex processing. Otherwise the vertex buffers of the mesh are not guaranteed to succeed in the rendering operation.

> *The current version of the* ITweenMesh *class expects you to supply all key-frames for a model in separate files. With most of key-framed file formats you will find all poses stored in a single file so loading those more complex files is obviously a task that is better done using the mesh factory. We will discuss this later in the chapter.*

1.7.3 Rendering a TweenMesh Object

Before we dig into the rendering function that does the vertex tweening with the two models I would like to show you the function that you can use to render one of the models of the tweened mesh just as it is without tweening taking place. You just need to delegate the call to the plain mesh class:

```
bool CTweenMeshRDO::RenderKeyframe( unsigned int uiNum, bool bOpaque,
                                    bool bAlpha )
{
	if ( uiNum < (unsigned int)m_vMeshs.size() )
	{
		if ( m_vMeshs.at( uiNum ) )
		{
			return m_vMeshs.at( uiNum )->Render( bOpaque, bAlpha );
```

```
			}
			else
			{
				return false;
			}
		}
		else
		{
			return false;
		}
}
/*----------------------------------------------------------------------------------------------------------*/
```

Rendering the mesh with active vertex tweening is not much more difficult either. First you have to check whether hardware tweening is possible or not and switch to software vertex processing if necessary. Then you activate the vertex blending render-state to enable vertex tweening and set the tweening factor which should be between 0.0f and 1.0f. The final preparation step is then to get the vertex buffers of both involved meshes and the index buffer of any one of them and activate this three buffers for the render device along with the vertex declaration. The actual trick is to find the two meshes out of the arbitrary number of allowed key-frame meshes which should be used at the current time interval and animation sequence.

For the sake of simplicity I assume here that all key-frames in a CTweenMeshRDO instance belong to one animation sequence as opposed to multiple, different animation sequences. Then its easy to get the two frames neighboring a certain time step in the range of [0,1] with 0 being the first key-frame and 1 being the last one in the animation sequence.

Finally, to render the tweened mesh you have to start a loop then for all materials of the mesh. For each material you query the according attribute range structure and activate the concerned material and texture if there is any. Then you can call the render function then and go on to the next material. Note that this is where you need access to the CMeshRDO's attribute buffer because you need to rebuild the mesh's render loop here in this function. You cannot use the IMesh interface's render function because it does not support vertex declarations instead of a flexible vertex format aka FVF.

And here is the complete implementation:

```
bool CTweenMeshRDO::Render( float fTween, bool bOpaque, bool bAlpha )
{
	static D3DXATTRIBUTERANGE* spAttrib = NULL;
	static CMeshRDO *pMesh_A = NULL, *pMesh_B = NULL;
	static IDirect3DVertexBuffer9 *pVB_A = NULL, *pVB_B = NULL;
	static IDirect3DIndexBuffer9  *pIB = NULL;

	// invalidate current device object settings
	m_pDevice->SetActiveIndexBuffer( NULL );
	m_pDevice->SetActiveVertexBuffer( NULL );

	// tween ranges from 0.0-1.0 so find the two key-frames concerned
	float fRealTween = fTween * (float( m_vMeshs.size() - 1.0f ));

	unsigned int uiA = (unsigned int)floorf( fRealTween );
	unsigned int uiB = (unsigned int)ceilf( fRealTween );

	if ( uiA == uiB ) return RenderKeyframe( uiA, bOpaqueSubset, bAlphaSubset );
	else { pMesh_A = m_vMeshs.at( uiA ); pMesh_B = m_vMeshs.at( uiB ); }

	fRealTween = fRealTween - floorf( fRealTween );

	// check hardware capabilities
	if ( !m_bHardwareTweening && m_pDevice->IsHardware() )
	{
		m_pDeviceD3D->SetSoftwareVertexProcessing( TRUE );
	}

	// activate vertex tweening rende state
	m_pDeviceD3D->SetRenderState( D3DRS_TWEENFACTOR,
	                              *((DWORD*)&fRealTween) );
	m_pDeviceD3D->SetRenderState( D3DRS_VERTEXBLEND,
	                              D3DVBF_TWEENING );

	// get the data buffers from the direct3d mesh object
	pMesh_A->GetMeshD3D()->GetVertexBuffer( &pVB_A );
	pMesh_B->GetMeshD3D()->GetVertexBuffer( &pVB_B );
	pMesh_A->GetMeshD3D()->GetIndexBuffer( &pIB );

	// activate vertex declaration and buffers
	m_pDeviceD3D->SetVertexDeclaration( m_pVertexDeclaration );
	m_pDeviceD3D->SetStreamSource( 0, pVB_A, 0, sizeof(SVertex_XYZ_N_1) );
	m_pDeviceD3D->SetStreamSource( 1, pVB_B, 0, sizeof(SVertex_XYZ_N_1) );
	m_pDeviceD3D->SetIndices( pIB );

	// draw the indexlist
	for ( unsigned long i=0; i<pMesh_A->GetNumMaterials(); i++ )
```

```
	{
		spAttrib = pMesh_A->GetAttributes( i );

		if ( pMesh_A->GetMaterial(i) ) pMesh_A->GetMaterial( i )->Activate();
		if ( pMesh_A->GetTexture(i)  ) pMesh_A->GetTexture( i )->Activate( 0 );
		else m_pDevice->EnableTextures( false );

		m_pDeviceD3D->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
					0, spAttrib->VertexStart, spAttrib->VertexCount,
					spAttrib->FaceStart*3, spAttrib->FaceCount );
		m_pDevice->EnableTextures( true );
	}

	pVB_A->Release();
	pVB_B->Release();
	pIB->Release();

	// set renderstates back to default
	m_pDeviceD3D->SetRenderState( D3DRS_VERTEXBLEND, D3DVBF_DISABLE );

	if ( !m_bHardwareTweening && m_pDevice->IsHardware() )
	{
		m_pDeviceD3D->SetSoftwareVertexProcessing( FALSE );
	}

	return true;
}
/*----------------------------------------------------------------------------------------------------*/
```

At the end of this function don't forget to switch back to hardware vertex processing if this is available and was switched off because vertex tweening was not supported in hardware.

This implementation of key-framed animation is rather simple because it only supports a single animation sequence for each ITweenMesh *instance. In your own implementation you should add the functionality to support multiple animation sequences using an arbitrary selection of key-frames from the instance. For some file formats this will require you to read in a separate ASCII file containing the animation sequence data.*

1.8 Tween Meshes and the Mesh Factory

Back to our IMeshFactory interface. As you will remember from paragraph 1.3 this interfaces does not only offer a function to create an IMesh instance from a static model provided in a separate file. It also offers a function to create an ITweenMesh instance from a file:

```
virtual bool CreateTweenMesh( const std::string &sFile, ITweenMesh **ppOut )const=0;
```

You can see in the CMeshLoader class implementing this interface that there is indeed a starting point for the function body. But there is no derived file loader class that would implement the tween mesh loading from a file such as the CModelLoaderAC3D for static meshes for example. The bottom line is that you need to implement such a loader for yourself for each file format you want to support. Possible file formats that support key-frame animations are the following ones:

- Microsoft DirectX .x
- Discreet 3D Studio Max .3ds
- id Software .md2 and .md3

I already pointed out that you will need to add a few adjustments to the ITweenMesh interface and the CTweenMeshRDO class implementation to support multiple animation sequences. At the moment you would need to project the start and end frames to the [0,1] range of the tween factor if you

have stored key-frames of multiple animation sequences in this class. So its possible but a bit uncomfortable. While implementing your first derived CModelLoaderMD2 for example you will see all problems arising from the current CTweenMeshRDO class because they just pop up.

Another issue is that some model formats that have key-framed animations separate three or even four different parts of the model. The .md3 format for example divides a character model into a lower part with hip and legs, a middle part from the belt upwards to the arms, and an upper part with the head. Optionally, there is also a separate weapon model (or a tool, but which shooter makes good use of tools *g*). The main model file does then define an attachment or so called ***tag point*** where you need to place the weapon model. The tag point could also be moving with the key-frames like a weapon would move while the soldier is running for example.

The bottom line is that you really need to look into the main file formats used by the top notch video games out there. A lot of editors and tools support the most famous file formats such as .md2 and .md3. which also means that you can find a lot of free models using those formats all over the internet. You will also find that a lot of those different file formats use the same ideas such as splitting the model into three separate parts and have a forth model along with a tag point. That means you should find a way to let your ITweenMesh interface definition reflect this common type of key-frame animated models. For example, your CTweenMeshRDO implementation could internally use three arrays of CMeshRDO instances for the lower, middle, and upper body parts of the model. Additionally, it would need to load a static weapon model and display it together with the animated model if requested.

You can see that a comprehensive key-frame animation class requires a bit of work if it should be as easy to use for an application as possible. That is why I won't discuss a more advanced implementation here. You have everything you need to go ahead on your own from here. Just do it. **:-)**

1.9 Summing Up

In this chapter we covered so much ground that it is hard to summarize everything with just a few sentences. First of all, you should split the whole content from your chapter that is floating around your mind into three folders. The first thing we did is to look at how Direct3D supports batching a bunch of triangles together into a so called mesh object. The second folder is labeled file format and introduced you to the way 3D artist store their work that must be loaded and rendered with a 3D engine. Finally, the third folder discussed vertex tweening and how this can be used to implement key-framed animation.

One other important concept you learned is the use of vertex declarations instead of the flexible vertex format. Actually, most vertex definition you can build with a vertex declaration can also be expressed using a flexible vertex format description. But using vertex declarations can be required for example for vertex tweening or when using a vertex shader. So you should look into those vertex declarations again reading the DirectX SDK documentation.