Graphics Programming Part I Appendix

The IWF SDK Overview

The IWF file format can seem complex at first if you are not used to working with 3D file formats. Although GILES natively exports its files to IWF format, their usage is in no way restricted to GILES. In fact, IWF plug-ins for some of the more popular modeling packages are available. GILES can also import 3ds files (models made in 3D Studio MAX) and X files (DirectX native format) and export them as IWF files.

Explaining the details of the IWF specification is beyond the scope of this course. Fortunately, we will not have to do so. The IWF SDK is included with this course and contains a 90+ page manual explaining the IWF file format in complete detail for those who are interested in creating their own plug-ins or low level file code. The SDK also ships with a lib and some header files that can be linked with your project to provide your application with helper classes to easily load the contents of an IWF file. The SDK includes a high-level import class that completely extracts the data from the file for you automatically into its internal structures. All your application needs to do is extract the mesh data from these structures and copy them into your own classes.

When you extract Lab Project 5.2 you will see that there is a folder called ***Lib***. This folder contains the IWF SDK library and header files that will provide your application with this high-level functionality. The IWF SDK documentation is included in PDF form in the ***Docs*** folder. In this course, we will be using the IWF SDK lib to automate the loading process for us.

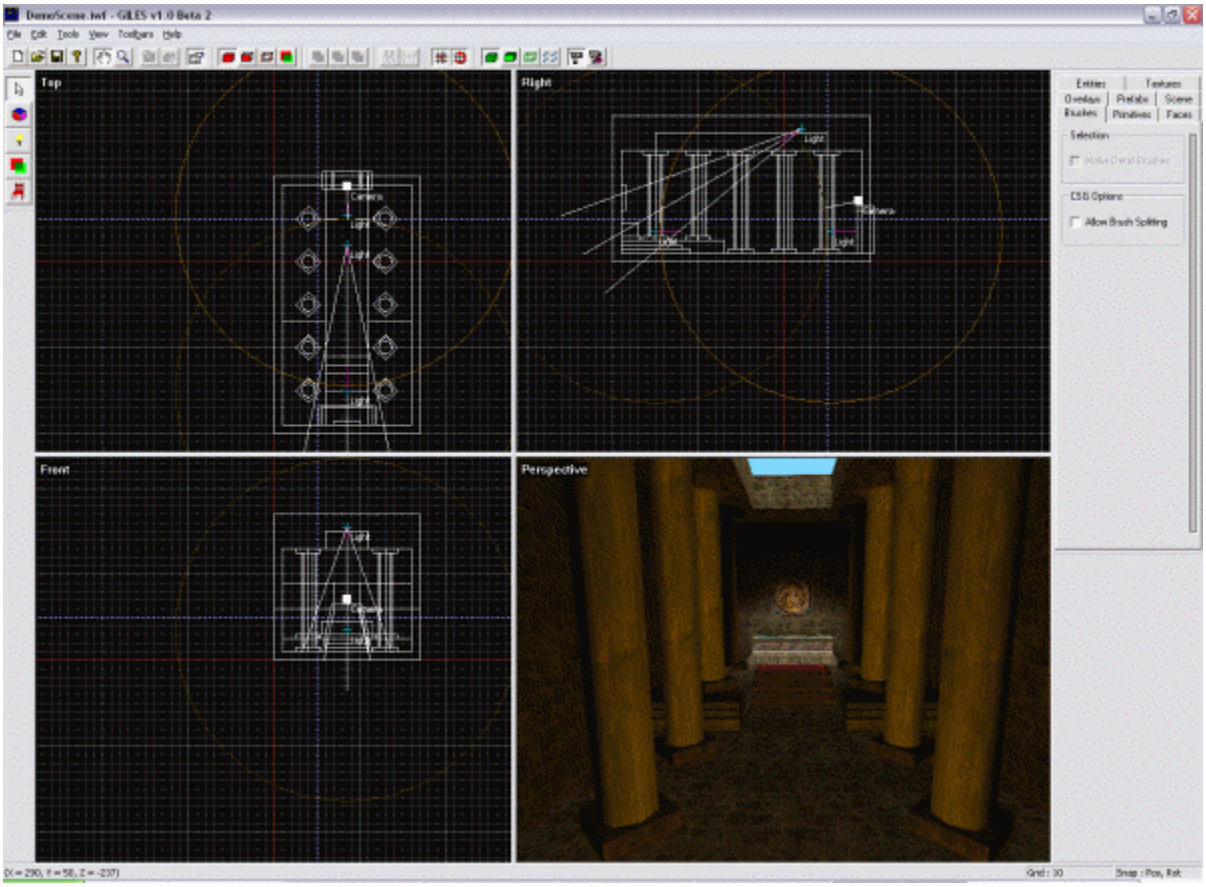
There are three **.h** files and one **.lib** file making up the SDK. Their names are:

iwfFile.h
iwfObjects.h
libIWF.h
libIWF.lib

The Lib file should be linked with your project (as we have done in Lab Project 5.2) and the header files should be included in the source code files that access the functionality.

GILES – Game Institute Level Editing System

GILES is a level editing system that helps you to build 3D worlds. It is especially useful for creating quick data for you to test with your engine. The level that we will be rendering in this next demo application was created using GILES. A copy of GILES is included free of charge with this course along with detailed documentation showing you how to use all of its features. A short tutorial on how to create your first 3D room is incorporated in the documentation for those of you that are new to using 3D world editors.



About the Level Data

GILES is a brush-based editor. This means that the world is constructed from very simple building blocks called **brushes**. A basic brush might be a cube or a cylinder or a sphere. These brushes can be combined into more complex brushes. A brush in GILES is what we might call a mesh. Each brush is made up from a number of faces, where a face is an N-sided polygon made up of one or more triangles. A hexagon for example could be a single face made up of many triangles. In the above image, each cylindrical pillar is a mesh, and the room itself started out as nothing more than a hollowed out cube mesh.

In GILES, materials and textures are applied at the face level, not the triangle level. Therefore, each face (which may be a collection of triangles) can have its own unique material or texture applied to it. Although it is very flexible for each face in the world to have its own material and texture, this can cause problems when rendering. As mentioned in Lesson Five, it is very important that we batch render our polygons. One might think that we could render all the polygons belonging to a single mesh in one go, but that will not be possible if the faces of that mesh have different materials or textures applied because we will need to change the device object's material before rendering these faces.

Have no fear. We will implement a system that decouples the actual rendering of polygons from the meshes to which they belong so that faces that share the same materials but belong to different meshes can be rendered together. We will basically collect the polygons from the meshes and assign them to vertex buffers that contain other polygons (possibly from other meshes) that share the same material. When rendering the scene we will set the device material that is associated with that vertex buffer and then render the entire contents in one go. Of course, it is worth pointing out that if you can avoid this situation, you should. If the polygons in a given mesh can all share the same properties, then that is preferable. We will revisit these issues later in the lesson as well as throughout the overall training program.

IWF File Format

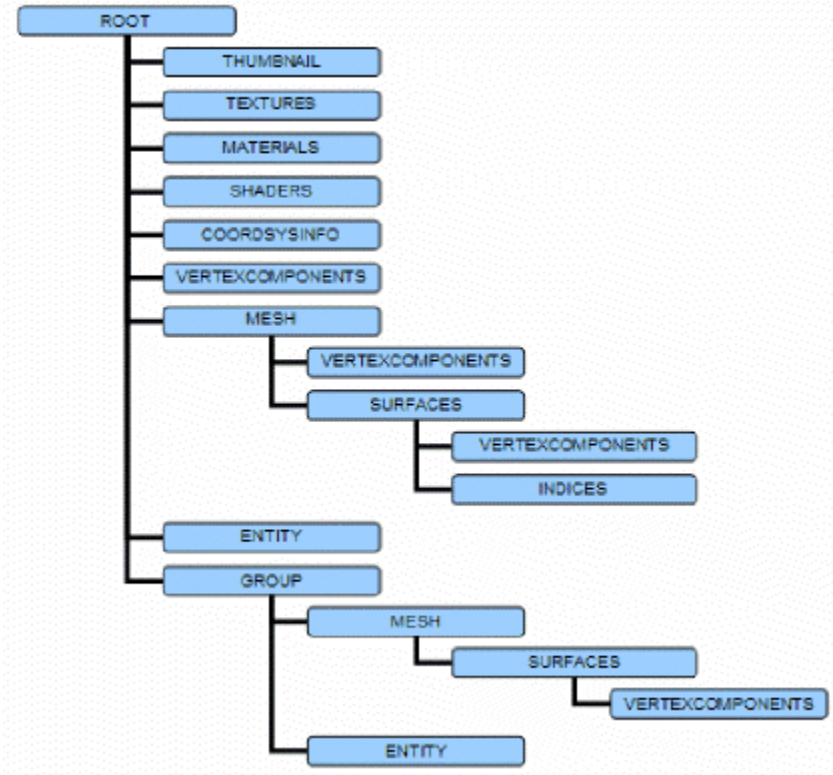
The IWF file is a hierarchical chunk format. This means that its components are stored in chunks of information. Each chunk has an ID. While third party IWF writers can create their own chunk types and chunk names, the standard reserved chunk names are defined in the file LibIWF.h as shown below:

```
#define CHUNK_ROOT                  0x0000
#define CHUNK_FILESUMMARY           0x0001
#define CHUNK_CONTAINER             0x0002
#define CHUNK_THUMBNAIL             0x0003
#define CHUNK_GROUP                 0x0010
#define CHUNK_MESH                  0x0020
#define CHUNK_DECALMESH             0x0021
#define CHUNK_SURFACES              0x0030
#define CHUNK_INDICES               0x0040
#define CHUNK_VERTEXCOMPONENTS      0x0050
#define CHUNK_ENTITY                0x0080
#define CHUNK_MATERIALS             0x0090
#define CHUNK_TEXTURES              0x00A0
#define CHUNK_SHADERS               0x00B0
#define CHUNK_COORDSYSINFO          0x00C0
```

You can think of each chunk as a block of information in the file. If the scene contained three meshes then the file would have three mesh chunks. Each one of these mesh chunks would have a surfaces chunk which describes the faces of that mesh. Each surfaces chunk would contain a vertices chunk describing the vertices used by each of the faces, and possibly an indices chunk as well. Many of these chunks are not used by geometry exported by GILES. Therefore, loading the file consists of loading

each chunk found in the file along with its child data. The IWF SDK documentation contains a complete list of all of the chunks and the data contained within each chunk.

The basic layout of an IWF file is shown below. A file will often contain only some of these chunks. If you have written your own IWF file writer, then it may contain additional chunks created and named by yourself solely for the purposes of your application. If you do use your own chunk names and layouts remember that third party IWF loaders will not be able to understand your custom chunk data.



Common Chunks

THUMBNAIL

If a THUMBNAIL chunk exists in the file it will contain a bitmap image of the contents of the file. This is used in GILES for example to show a preview of the file. When GILES saves an IWF file, it will take a screen shot of the level as shown in the 3D view and will save this bitmap in the thumbnail chunk of the file. This means the IWF loader dialog within GILES can give the user a graphical preview of the file without the user having to load it.

TEXTURES

The TEXTURES chunk contains all of the textures used by the level. This chunk may contain the actual texture pixel data or a list of texture names which can be used as a file name to load the file externally. The texture chunk in a GILES-created IWF file contains a table of texture names. The texture files themselves are external to the file and must be loaded separately. Lab Project 5.2 will not use textures and therefore this chunk will not exist in the file. However Chapter 6 will contain a Lab Project that makes use of this chunk.

MATERIALS

The MATERIALS chunk contains all of the material data used by the scene. Internally, each material is stored in this chunk in a format much like the D3DMATERIAL9 structure. Each material has an ambient, diffuse, and specular member. Additionally, each material can have a name associated with it. Remember that like the textures chunk, there is usually only one material chunk per file stored in the root of the file. Each mesh face (i.e. surface) has an index into this material list describing the material the surface uses. In GILES every face has exactly one material applied. This means that inside the IWF file exported by GILES, every surface contains one material index between zero and the number of materials in this global material list.

SHADERS

The SHADERS chunk provides support for stored vertex and pixel shaders as well as effect files. We will not be using these concepts until much later in the training program, so this chunk will not exist in the IWF file that we will be using in Chapter5Demo2.

COORDSYSINFO

The COORDSYSINFO chunk allows the file creator to store a 4x4 matrix. The loading application can multiply this matrix with each vertex in the file to convert them into a standard left-handed Cartesian coordinate system (as used by DirectX Graphics). GILES naturally stores geometry using the left-handed coordinate system so this matrix is not needed and is not contained in the file.

VERTEXCOMPONENTS

The VERTEXCOMPONENTS chunk will usually exist as a child chunk of the SURFACES chunk (this is always the case with IWF files created by GILES). When this chunk exists in the root, the indices of every face (or every mesh) in the file will index into a global vertex pool stored in the root. This means all meshes index into the same large array of vertices within the file. When this chunk exists as a child of the mesh (instead of at the root), each mesh will have its own list of vertices (vertex pool) and the indices of each face belonging to that mesh index into this vertex pool. Only this mesh's faces use this vertex pool so every mesh in the file will have its own VERTEXCOMPONENTS chunk. Finally, the file may instead store a VERTEXCOMPONENTS chunk as a child of each surface. This

means each surface (face) will have its own pool of vertices that only it uses. If this is the case, the indices of each face will always be relative to its own vertex pool.

GILES uses the 3rd technique of those listed above. Each mesh in the file contains a surface chunk which describes the properties of each face (which material it uses from the material list for example) and each surface contains a VERTEXCOMPONENT chunk containing an array of vertices belonging to that face. Therefore, the indices stored for each face (also as a child of the surface chunk) are always zero based where index=0 is the first vertex in this surfaces vertex list and index=NumberOfVerticesInFace-1 describes the last vertex in the surfaces vertex pool.

MESH

A MESH chunk will exist for every separate mesh in the file. We discussed in Workbook Chapter 5 how GILES stores the file as a series of meshes. Each separate brush in GILES is saved out as an IWF mesh with its own mesh chunk within the file. If you made a level in GILES that consisted of 300 brushes, there would be 300 separate mesh chunks in the IWF file. Each MESH chunk contains information about the mesh, such as its name and possibly a world matrix that should be used to transfer the brush into world space. GILES does not store this matrix in the mesh chunk since all vertices are defined in world space by default. Each mesh chunk will also have child chunks, and every mesh must have at least one SURFACES chunk as a child which describes the faces (surfaces) of that mesh. If the file contained a vertex pool for each mesh, then the MESH chunk would also have a VERTEXCOMPONENTS chunk as a child. GILES does not use mesh vertex pools and instead stores per-face vertex pools as a child of each SURFACES chunk.

SURFACES

The SURFACES chunk will always exist as a child chunk of the MESH chunk. Each mesh has one SURFACES chunk which contains all of the information about the surfaces (faces) making up the mesh. A cube mesh for example would have one SURFACES chunk containing data for the six surfaces making up the cube. Each SURFACES chunk contains information about the materials and textures used by each surface in the mesh. These are stored as indices into the TEXTURES and MATERIALS chunks.

INDICES

The INDICES chunk may or may not exist as a child of a SURFACES chunk depending on whether or not the surfaces stored in that chunk are stored as vertex lists (triangle lists, triangles fans, or triangle strips) or whether the surfaces are stored as indexed lists (indexed triangle lists, indexed triangle fans or indexed triangle strips). If the surfaces stored are indexed primitives, then an indices chunk will exist and contain the index lists for each surface in the list.

The INDICES chunk can be stored as a child of each SURFACES chunk. If the SURFACES chunk contained 10 surfaces for example, then the indices chunk would contain 10 lists of indices used to describe each surface as a series of triangles. In GILES, these indices index into the vertex pool stored per SURFACES chunk.

ENTITY

The ENTITY chunk is used to store objects that are not meshes. Entities can be pretty much anything. Entities plug-ins can be created by third parties and added to GILES. GILES ships with only two Entities, the LIGHT entity type and the LOCATION entity type. An entity from the level editor point of view is a block of information input by the user and linked to a location in 3D space inside the editor. Each entity placed in GILES has a location and orientation in the 3D world plus a collection of other data input by the user.

For example, lights are stored as entities in GILES. Each light entity has a position and an orientation that is assigned to the entity when the user places it within the world. The user can select the entity and fill out details specific to that entity. In the case of a light entity, this additional information contains fields such as ambient, diffuse, and specular colors, range, attenuation, and cone angles (for spotlights only) and information describing whether the light is a spotlight, a point light, or a directional light. If you have ever placed lights within GILES then this will be instantly familiar to you. You might also notice that GILES lights are very similar to DirectX Lights as described by the D3DLIGHT9 structure.

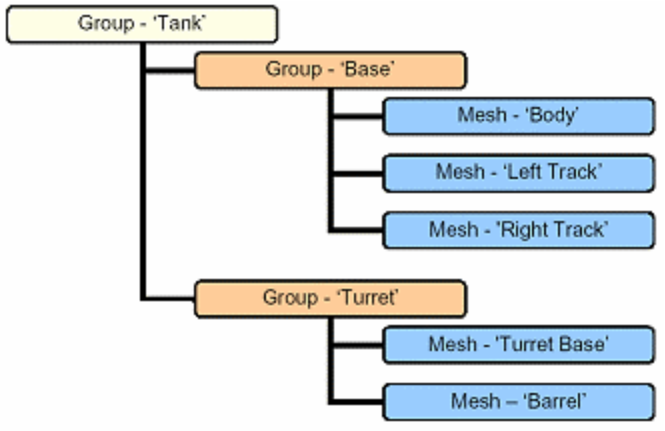
When the IWF SDK helper class loads an IWF file, it will check each entity to see if it is a light entity. Each entity chunk will store an identification known as an ENTITYTYPE_ID. The current standard reserves the IDs of only three entity types. These IDs are defined in LibIWF.h as shown below:

```
#define ENTITY_LIGHT              0x0010
#define ENTITY_SPAWN_POINT        0x0030
#define ENTITY_SIMPLE_POINT       0x0031
```

GROUP

The GROUP chunk allows us to group objects (meshes, entities etc) together into logical orders to construct a scene hierarchy. In the file itself, a group is nothing more than a chunk that contains a string naming the group. Each group can have child chunks (meshes, entities or other groups for example) and any child groups can also contain child chunks of their own. This allows us to store the scene as a scene graph where objects of relevance are stored and grouped by name.

As an example, imagine we had used GILES to create a tank. The tank could consist of multiple meshes, perhaps a mesh for its turret and another mesh for the gun of the turret and also perhaps another three meshes used to create the main body and the left and right tracks. When the tank moved, we would want all of the meshes making up the tank to be moved together. Therefore, we would assign all of these meshes to the same group; perhaps we would call this group ‘Tank’. We may also decide that the turret should be able to be rotated independently from the tank body. So we could create two more child groups to contain the meshes forming the turret (the turret base and the gun) and another group containing the meshes making up the main body of the tank as shown in the diagram below.



Using groups, we know that if the user rotates the turret 45 degrees left, we can build the rotation matrix and apply it to all the meshes in the 'Turret' group. In that case, only the turret rotates and not the entire tank. If we wish to rotate the entire tank, then we can rotate the Tank group itself and thus every mesh which is a child (either directly or indirectly) will be rotated in turn.

Loading IWF Files with the IWF SDK

The IWF SDK contains two loader classes to assist applications with loading IWF files. The first of these classes is declared in libIWF.h and is called CBaseIWF. This class is a low-level helper class that provides functions for reading, writing, and identifying chunks. You will derive from this class to create your own file loading classes that will be responsible for loading the data from each chunk within the file. Using this class requires a thorough understanding of the IWF SDK and you should not try to use this class unless you have read the IWF SDK documentation. Because this class is so low-level, we will not be using it directly in this course. The IWF SDK also contains a much higher-level class called CFileIWF (declared in IWFFile.h). This class is derived from CBaseIWF and does all of the grunt-work for you. This class loads all of the data automatically with a single function call. This is the class that our applications will use. The following lines of code show you how easy it is to load an IWF file using this class. First we create an instance of the CFileIWF class and then we call its Load member function passing in the name (and the path, if necessary) of the file as shown below.

```
#include "iwfFile.h"
…
…
CFileIWF   File;
File .Load("Example.iwf")
```

After the call completes, all of the objects in the IWF file will have been loaded into memory and stored within arrays maintained by the CFileIWF class. All that has to be done now is to query CFileIWF object for the number of meshes, entities, surfaces, etc. and extract the data from its member arrays.

The CFileIWF class definition is listed below. We have only shown the public variables and methods of the class since these are all our application needs to be aware of.

```
class CFileIWF : public CBaseIWF
{
public:
    // Constructor and Destructor
             CFileIWF();
    virtual ~CFileIWF();

    // Public Functions for This Class.
    void            Load( LPCTSTR FileName, ULONG Flags = 0 );
    void            ClearObjects();


    // Public Variables for This Class.
    vectorIWFMesh      m_vpMeshList;       // A list of all meshes loaded.
    vectorIWFTexture   m_vpTextureList;    // A list of all textures loaded.
    vectorIWFMaterial  m_vpMaterialList;   // A list of all materials loaded.
    vectorIWFEntity    m_vpEntityList;     // A list of all entities loaded.
    vectorIWFShader    m_vpShaderList;     // A list of all shaders loaded.
    vectorIWFGroup     m_vpGroupList;      // A list of all groups loaded.
};
```

The member variables are all STL vectors defined in the same file as follows:

```
typedef std::vector<iwfGroup*>      vectorIWFGroup;
typedef std::vector<iwfMesh*>       vectorIWFMesh;
typedef std::vector<TEXTURE_REF*>   vectorIWFTexture;
typedef std::vector<SHADER_REF*>    vectorIWFShader;
typedef std::vector<iwfMaterial*>   vectorIWFMaterial;
typedef std::vector<iwfEntity*>     vectorIWFEntity;
```

The STL vectors contain a list of all the groups, meshes, textures, shaders, materials, and entities in the file. When we call the CFileIWF::Load function, the class opens the file, loads all the data, and adds it to the above vectors. Because these member vectors are public, we can easily access them from our application after the Load function has returned. In the following example code, we retrieve the number of elements in the various vectors:

```
CFileIWF File;

// Load iwf file
File.Load("Example.iwf");

// Retrieve number of meshes loaded
long MeshCount     = File.m_vpMeshList.size();

// Retrieve number of materials
long MaterialCount = File.m_vpMaterialList.Size();

// Retrieve number of entities
long EntityCount   = File.m_vpEntitylList.Size();

// Retrieve number of textures
long TextureCount  = File.m_vpTextureList.Size();
```

All of the proprietary IWF classes and structures are listed below and are part of the IWF SDK. They are declared in iwfObjects.h. The structures used by the CFileIWF class to store objects loaded from an IWF file are:

iwfGroup
iwfMesh
iwfMaterial
iwfEntity
TEXTURE_REF
SHADER_REF

We will now look at each of these data types one at a time so that we know how to access the data we need.

iwfMesh

The CFileIWF::m_cpMeshList vector contains an array of meshes that were found in the IWF file. Each mesh is stored in the vector as an iwfMesh structure:

```
class iwfMesh : public IIWFObject
{
public:

// Constructor / Destructor
    iwfMesh( ULONG Count );
    iwfMesh();
    virtual ~iwfMesh();

// public member functions
    UCHAR       QueryType( ) { return OBJECT_TYPE_MESH; }
    long        AddSurfaces( ULONG Count = 1 );

// public member variables
    char       *Name;                   // The mesh name itself
    ULONG       Components;             // Which components are valid
    MATRIX4     ObjectMatrix;           // The mesh's transformation matrix
    AA_BBOX     Bounds;                 // The mesh's world space bounding box
    ULONG       Style;                  // Mesh's Style
    USHORT      CustomDataSize;         // The size of the custom data area
    UCHAR      *CustomData;     // The data stored in the mesh custom data area
    ULONG       SurfaceCount;   // Number of polygons stored
    iwfSurface**Surfaces;       // Simply surface array.
};
```

Each mesh contains a name and a series of flags describing which components are valid. The mesh can also contain a bounding box describing a box which surrounds the mesh geometry (to be used for collision detection or visibility culling). The mesh may also contain a world matrix that has been imported from the file. If so, this is stored in the **ObjectMatrix** member. The Mesh may have its style member set to something other than zero which informs the application about what type of mesh this was designed to be used for. This bit set can be zero or can have the following flags set (defined in LibIWF.h).

```
#define MESH_DETAIL                 0x1
#define MESH_DYNAMIC                0x2
#define MESH_DESTRUCTIBLE           0x4
```

None of these flags affect the geometry in any way. They are a means for the level designer to inform the application about how the meshes should be treated. For example, if we wanted to compile our level geometry into a BSP tree (we will examine BSP trees later in the curriculum), we typically only want to compile the basic geometry into the tree. For reasons that are beyond the scope of this course, it is unwise to compile high detail objects (such as highly tessellated spheres) into the tree. In GILES, the level designer can flag these high polygon meshes as detail brushes which will be exported with the MESH_DETAIL flag set. When our BSP compiler processes this data it can check for meshes with this flag set and not compile their geometry into the BSP tree. The MESH_DYANMIC flag might be set by the level designer to inform the application that this mesh will move around a lot in the level, so perhaps should not be stored in a static vertex buffer. Finally the MESH_DESTRUCTIBLE flag could be set by the level designer to inform the application which meshes are allowed to be destroyed by the player. Most of the levels we use in this course will have none of these flags set.

The Mesh also contains a custom data area. This can be literally anything the IWF file creator wants to store in the file to be associated with this mesh. GILES does not use this custom data area so it will not exist in IWF files exported from GILES.

To know which of these members a given mesh supports we look at the **Components** member of each mesh. This is a bit field that describes which, if any, of the following members are used by the mesh. These flags are also defined in libIWF.h.

`#define MCOMPONENT_OMATRIX 0x1`
The ObjectMatrix member of this mesh is valid and contains a world matrix.

`#define MCOMPONENT_BOUNDS 0x2`
The Bounds member of the mesh contains a valid axis aligned bounding box (AABB). The Bounds member is of type AA_BBOX which is a structure containing two 3D vectors describing the minimum and maximum extents of the bounding box as world space positions. The structure is defined in iwfObjects.h and is shown below.

```
typedef struct _AA_BBOX
{
        // Public Variables for This Struct
       VECTOR3  Min;        // The minimum extents of the bounding volume
       VECTOR3  Max;        // The maximum extents of the bounding volume

        // Constructors
       _AA_BBOX( const VECTOR3& _Min, const VECTOR3& _Max )
       {
              Min = _Min;
              Max = _Max;
       }
```

```
        _AA_BBOX()
        {
              Min = VECTOR3(  9999999.0f,  9999999.0f,  9999999.0f );
              Max = VECTOR3( -9999999.0f, -9999999.0f, -9999999.0f );
        }
  } AA_BBOX;
```

If the MCOMPONENT_BOUNDS flag is not set then this mesh does not have its bounding box stored within the IWF file and you will need to calculate it yourself if desired. GILES always exports the bounding box of each mesh in the IWF file, so this flag will be set for all mesh contained in an IWF file exported from GILES.

#define MCOMPONENT_STYLE 0x4

If this bit of the component member is set then the style member is valid. This means that the mesh has been assigned a special property by the designer.

#define MCOMPONENT_CUSTOMDATA 0x8

If this bit its set then it means that the CustomData member pointer points to additional data associated with this mesh. The CustomDataSize describes the size of this custom data in bytes. This data area is not used by meshes exported from GILES but could be used by a third party IWF writer to store a mesh description for as GUI application, or perhaps if the mesh was a space ship, this data area could contain the speed, shield strength and maneuvering capabilities of the mesh.

#define MCONPONENT_ALL 0xF

If this bit of the components member is set then it means all of the above fields are valid and contain valid data.

The last two members of the iwfMesh class contain an array of iwfSurface structure pointers and a count describing how many surface pointers are in this array. If this was a cube for example, this array would contain an array of six iwfSurface structure pointers describing the six faces of the cube. The following code shows how we could find out how many faces (surfaces) exist in the sixth mesh of the CFileIWF class. This code assumes that the IWF file loaded contained at least six meshes.

```
#include "iwfFile.h"
#include "iwfObjects.h"

CFileIWF File;
File.Load("Example.iwf");

IwfMesh *MyMesh  = File.m_vpMeshList[5];
ULONG  FaceCount = MyMesh->SurfaceCount;
```

iwfSurface

The iwfSurface class represents a single face in a mesh. Each face is stored as an array of vertices and possibly an array of indices. Faces are stored in IWF files as N-sided polygons with a clockwise winding order. Each face can be broken down into multiple triangles. You can assume that GILES stores faces in the same way that we would store a triangle fan primitive. In chapter 3 of the course, we rendered a fan by passing an array of vertices into the DrawPrimitive function (defined in a clockwise winding order). The first three vertices in this array define the first triangle and then every additional vertex creates a new triangle using the current vertex, the previous vertex, and the first vertex in the list.

A face may or may not be able to be rendered successfully as a triangle fan. If this is the case then the face may contain an additional index list that describes the face as a series of indices into the vertex array. Only faces that need indices to render properly are saved with an index list in GILES (simply to save space). When an index list is available, the face is described as an indexed triangle list. Even if the face is stored as a fan originally, it is certainly no trouble to convert it to an indexed triangle list to achieve better rendering performance.

```
class iwfSurface
{
   public:

   // Constructors & Destructors for This Class.
    iwfSurface( USHORT VertexCount );
    iwfSurface();
    virtual ~iwfSurface();

   // Public Functions for This Class
    long        AddVertices( USHORT Count = 1 );
    long        AddIndices ( USHORT Count = 1 );
    long        AddChannels( UCHAR  Count = 1 );

  // Public Variables for This Class
    ULONG         Components;           // Which components are valid
    VECTOR3       Normal;               // Surface Normal
    UCHAR         ZBias;                // Surface ZBias
    ULONG         Style;                // Style flags
    SCRIPT_REF    RenderEffect;         // The effect to apply to this surface
    UCHAR         ChannelCount;         // Number of shader channels
    short        *TextureIndices;       // Simple array of texture indices
    short        *MaterialIndices;      // Simple array of material indices
    short        *ShaderIndices;        // Simple array of shader indices
    BLEND_MODE   *BlendModes;           // Simple array of blend modes
    USHORT        CustomDataSize;       // The size of the custom data area
    UCHAR        *CustomData;           // Mesh custom data area

    ULONG      VertexComponents;  // The components stored for each vertex
    UCHAR      VertexFlags;       // Various flags relating to the verts stored
    USHORT  VertexCount;     // Number of vertices stored
    UCHAR   TexChannelCount; // Number of vertex texture channels
    UCHAR  *TexCoordSize;    // How many dimensions each texture coordinate has

    UCHAR  IndexVPool;  // Vertex pool from to which the indices point
```

```
    UCHAR  IndexFlags;  // The format of the indices (i.e. INDICES_TRILIST)
    USHORT IndexCount;  // Number of indices stored.

    iwfVertex  *Vertices;  // Simple vertex array
    ULONG      *Indices;   // Simple index array (32 bits per index capacity)

};
```

You may be starting to notice some parallels here with the classes we used earlier in the course to store a mesh. We used our own CMesh class as a container for an array of CPolygon structures that contained the face information for the mesh. Each CPolygon contained a CVertex array which contained a list of vertices describing the face. This is all very similar. The iwfMesh contains an array of iwfSurfaces describing the information for each face in the mesh. Each iwfSurface contains an array of iwfVertex structures and an array of indices describing the triangles making up that face. The iwfSurface class however also includes a load of other members which may or may not be valid in the loaded mesh. Some of these members are used for advanced effects either not supported by our early demos and some not even supported by GILES.

The iwfSurface class has constructors and member functions to create the surface and add vertices to the surface. These member functions are used by the CFileIWF class to create the surfaces when loading the file, but you can still use these classes to maintain your own geometry even if you are not loading an IWF file. In this case, you will need to use these functions to create surfaces and add vertices to these surfaces. The CFIleIWF class creates all surfaces for us automatically, so we are only interested in the member functions at this point and not how to create or amend faces.

ULONG Components;

The components member of the iwfSurface class describes which member variables contain valid information. This is important because not all surfaces will contain valid information in all of the fields of the iwfSurface members. For example, GILES does not export the RenderEffect property and does not export any custom data for each face. So both the RenderEffect and CustomData members will not be valid in IWF files created by GILES. This member is a bit field (32 bits) which can have one or more of the following bits set (defined in libIWF.h):

```
#define SCOMPONENT_NORMAL           0x1
#define SCOMPONENT_ZBIAS            0x2
#define SCOMPONENT_EFFECT           0x4
#define SCOMPONENT_STYLE            0x8
#define SCOMPONENT_TEXTURES         0x10
#define SCOMPONENT_MATERIALS        0x20
#define SCOMPONENT_SHADERS          0x40
#define SCOMPONENT_BLENDMODES       0x80
#define SCOMPONENT_CUSTOMDATA       0x100
#define SCOMPONENT_ALL              0x1FF
```

The following code shows how we could test to see if the fifth surface of the mesh has valid face normal information:

```
D3DXVECTOR FaceNormal;
IwfSurface   *pFace = MyMesh->Surfaces[4];
If (pFace->Components & SCOMPONENT_NORMAL) FaceNormal = pFace->Normal;
```

VECTOR3 Normal;
A unit length 3D vector describing the orientation of the face.

UCHAR ZBias;
ZBias is useful for eliminating rendering artifacts that can occur when two polygons are rendered with the same position and orientation in the 3D world (overlapped). Imagine a wall polygon in the game world with a brick wall texture to it. The level designer may want to create another polygon with a bloodstain texture for example and then lay it over the wall face (so that it looks like the blood stain is on the wall). The problem here is that these faces share the exact same 3D space and the Z Buffer will have trouble when rendering the second of these two surfaces. Let us assume that the wall polygon was rendered first and its depth values are recorded in the Z Buffer. When we try to render the bloodstain polygon we find that it shares the same depth information because it is exactly the same distance from the camera. In this case, the bloodstain will either not be rendered at all because it is rejected by the depth buffer, or worse (because of floating point inaccuracies incurred during per-pixel interpolation), some of the pixels of the blood stain may pass the depth test and some may not. This causes a flickering effect where only fragments of each polygon are rendered.

Modern 3D APIs such as DirectX 9 contain a ZBIAS function which instructs the device to marginally offset the second polygon by some small amount so that it successfully renders on top of the first polygon. The IDirect3DDevice interface exposes a render state called D3DRS_ZBIAS which allows you to set the current ZBias used by the renderer. The range of values is [0, 16]. First you set the ZBias to zero for normal rendering (this is the default ZBias for a newly create device):

```
m_pDevice->SetRenderState( D3DRS_ZBIAS , 0);
```

Then you render the wall polygons. Next we wish to render the bloodstain polygon but we want it to appear just in front of the wall polygon by a tiny amount. Polygons rendered with a higher ZBias value appear in front of polygons rendered with a lower ZBias value. Therefore, we could set the ZBias to 1 and then render the bloodstain polygon:

```
m_pDevice->SetRenderState( D3DRS_ZBIAS , 1);
```

We now set the ZBias back to zero in preparation for rendering any other non-overlapped polygons. The ZBias does not alter the position of the polygon in the 3D world. It just biases the depth test such that it makes each pixel appear closer in the depth buffer than non-biased pixels.

ZBias as well as its value range differs from API to API. To accommodate this, the IWF file stores ZBias values in the range [0, 255]. You can scale this value into the range for DirectX by dividing the ZBias values stored in the iwf File by 255 and multiplying the result by 16. So if the IWF file contained a ZBias of 128 this would generate a Direct X compatible ZBias value of:

128 / 255 = 0.50196
0.50196*16 = (int) 8

Support for DirectX ZBias is pretty shaky across drivers so it really should not be used until better support becomes available in the future. But all is not lost. We can convert this value into an alternative means to manipulate the projection matrix to obtain the same result in a hardware independent way. We will cover this technique later in the curriculum.

ULONG Style;
Each face in GILES can also be assigned one of four reserved flags which describe the face as being special. These flags are defined in libIWF.h and are shown below along with a description of their meaning.

#define SURFACE_INVISIBLE 0x1
If this flag is set it means that the level designer intends this face to be invisible. This usually equates to it not being rendered. It may seem strange to think of a face being in a file that is not to be rendered but there are many practical uses for this. One example would be collision detection. A level designer may place an invisible face to create an invisible force field that the player cannot walk through. Perhaps cubes might be placed to mark off different areas of the level. When the player enters this invisible cube the game might spawn an evil monster.

#define SURFACE_FULLBRIGHT 0x2
Sometimes a level designer may want a face to be fully lit regardless of whether the face is within range of a light.

#define SURFACE_SKYTYPE 0x4
It is often useful for the level designer to inform the application which faces have been used to form the sky of the world. This might be useful if the application wanted to animate its cloud texture to give the impression that the clouds are moving.

#define SURFACE_TWO_SIDED 0x8
This flag indicates that the face should be rendered without backface culling.

SCRIPT_REF RenderEffect;
This member contains information about a script that is associated with this face. This might be used to trigger a scripted event such as a cutscene when the player collides with this face. Scripts are not supported by GILES and are beyond the scope of this course.

UCHAR ChannelCount;
A single face can have more than one material and/or texture applied to it. A surface in the IWF file can contain a number of channels, where each channel can consist of a texture and a material. If either the material or texture indices for that channel are set to –1 then it means that the channel does not use a material or texture respectively. All surfaces in GILES have a material applied to them and there will always be a channel count of (at least) one and one material index preface. This material describes the material for channel count 1. If the surface has a texture applied as well, then the texture index will be stored in the first element in the texture indices list. You can think of each channel as being the number of passes needed to render a face without single pass multi-blending.

short *TextureIndices;
This is an array of texture indices indicating the texture index per channel. Each index references into the IWF file texture table. In the case of GILES, this refers to a list of texture file names stored in the textures chunk. Each element corresponds to the channel with the same index. The value will be –1 if the face has no texture(s) applied.

short *MaterialIndices;
This is an array of material indices stored in the CFileIWF materials vector. There is exactly one material per channel. This value will not be –1 for GILES levels since every face in GILES must have a material assigned to them. Remember that the CFileIWF class has loaded all the materials found in the file into its materials vector so this index can be used to retrieve the material from that array.

short *ShaderIndices;
This member contains a list of indices into the shaders list stored in the file. We will not be loading shaders from the IWF files in this course since the shaders stored will usually be application defined. We will be covering shaders later in the curriculum.

BLEND_MODE *BlendModes;
The DirectX API gives us the ability to render polygons in such a way that instead of overwriting any contents previously stored in the frame buffer, the polygon is blended with the current contents of the frame buffer. This is called alpha blending and there are several different blending equations that can be used to produce different effects. Alpha blending is discussed in detail in chapter 7 of this course. Blending effects are achieved by setting two blending ratios. These determine the final color of each pixel rendered by using a mixture of the two input colors. The input colors to this equation are the color of the pixel about to be rendered and the pixel color already in the frame buffer. The BLEND_MODE structure is shown below:

```
typedef struct _BLEND_MODE
{
    UCHAR       SrcBlendMode;               // Source Blend Mode
    UCHAR       DestBlendMode;              // Destination Blend Mode
} BLEND_MODE;
```

Both of these values can be assigned one of the values stored in the following table.

Name	Value	Description
BLEND_NONE	0x0	No blending is to be applied.
BLEND_ZERO	0x1	Blend factor is (0, 0, 0, 0).
BLEND_ONE	0x2	Blend factor is (1, 1, 1, 1).
BLEND_SRCCOLOR	0x3	Blend factor is (R_s, G_s, B_s, A_s).
BLEND_INVSRCCOLOR	0x4	Blend factor is (1–R_s, 1–G_s, 1–B_s, 1–A_s).
BLEND_SRCALPHA	0x5	Blend factor is (A_s, A_s, A_s, A_s).
BLEND_INVSRCALPHA	0x6	Blend factor is (1–A_s, 1–A_s, 1–A_s, 1–A_s).
BLEND_DESTALPHA	0x7	Blend factor is (A_d, A_d, A_d, A_d).
BLEND_INVDESTALPHA	0x8	Blend factor is (1–A_d, 1–A_d, 1–A_d, 1–A_d).
BLEND_DESTCOLOR	0x9	Blend factor is (R_d, G_d, B_d, A_d).
BLEND_INVDESTCOLOR	0xA	Blend factor is (1–R_d, 1–G_d, 1–B_d, 1–A_d).

USHORT CustomDataSize;
UCHAR *CustomData;
These two members are used to describe the size of a custom data area (in bytes) associated with this face as well as the custom data itself. Third party level editors may allow the level designer to store additional data with each face (GILES does not). These members will not be used by any of our applications in this course.

ULONG VertexComponents;
This value describes the vertex format. The vertices may contain a normal for example or may contain a diffuse and specular color. If the face has a texture applied to it then it will also have at least one set of texture coordinates describing how the texture is mapped to the surface. Our applications can use these flags to make sure that it creates a vertex buffer capable of holding the correct FVF vertex type. Below are the flags that may or may not be set in the VertexComponent bit field (defined in libIWF.h).

#define VCOMPONENT_XYZ 0x1
This flag will always be set with files created in GILES since each vertex will have an X, Y, and Z position component. Although this will usually contain the object space position of the vertex, in GILES all vertices are exported in world space. To render these faces we simply set the device world matrix to an identity matrix so that the world transformation is not applied.

#define VCOMPONENT_RHW 0x2
#define VCOMPONENT_XYZRHW 0x3
These vertices have X,Y, and Z position components expressed as screen space positions (X and Y) and a pre-calculated depth value (Z). These polygons are not pumped through the transformation and lighting pipeline. RHW stands for *reciprocal of Homogeneous W* and contains the 1/W calculation usually done by the pipeline during transformation. This value

must be specified so that fog effects and W buffers work correctly with this type of 2D vertex. Pre-transformed vertices are useful for rendering 2D items using 3D hardware acceleration. We will discuss these vertex types later in the curriculum. GILES does not export this type of vertex.

#define VCOMPONENT_NORMAL 0x4
This flag indicates that the vertices used by this face include vertex normals.

#define VCOMPONENT_POINTSIZE 0x8
This flag is used if the vertex is being used to represent the point sprite primitive type. Point sprites are special cases of points that enable us to bypass the one-pixel size limitation imposed by DirectX. We will discuss point sprites later in the curriculum.

#define VCOMPONENT_DIFFUSE 0x10
If this flag is set then it means each vertex of this face contains a diffuse color.

#define VCOMPONENT_SPECULAR 0x20
If this flag is set then it means each vertex of this face contains a specular color.

#define VCOMPONENT_TEXCOORDS 0x40
If this flag is set it means the TexChannelCount member of the iwfSurface contains the number of texture coordinates in each vertex. A single surface can have multiple textures blended and applied to it. If this is the case then the vertex may contain multiple sets of texture coordinates describing how each individual texture maps to the face.

#define VCOMPONENT_TEXCOORDLAYOUT 0x80
If this flag is set then it means the TexCoordSize member pointer of the iwfSurface will point to an array of texture coordinate sizes. Usually, texture coordinates are 2D coordinates describing how the vertex maps to a 2D pixel in the texture (pixels in textures are called texels). It is also possible to have 1D, 2D, or even 3D texture coordinates and the TexCoordSize array will contain the dimension size of each set of texture coordinates in the vertex. Texture coordinates are covered in detail in chapter 6 of this course.

#define VCOMPONENT_ALL 0xFF
If this flag is set it is the equivalent to all of the above flags having been set.

UCHAR VertexFlags;
This member presents the loading application with information about the order in which the vertices are stored. This member will have one (or sometimes more) of the following bits set indicating the vertex order (defined in libIWF.h):

#define VERTICES_NON_PLANAR 0x1
If this bit is set then it means that the vertices may not all be coplanar. For example, the outside of cylinder could all be stored as one surface consisting of many triangles each on a different plane. It is important to know this information so that loading applications do not try and use one face normal to represent the orientation of this surface. If the vertices are non-planar then

the surface orientation cannot be represented with a single face normal and the loading application may have to generate a face normal for each triangle making up the surface. This flag must be combined with one of the following flags. GILES does not export non-planar vertex pools so this flag will never be set for vertex pools exported from GILES.

#define VERTICES_INDEXED 0x2

If this flag is set it means that the vertices are not stored in any particular rendering order. This surface will include a list of indices that describe the triangles of this surface from the vertex list. GILES usually exports surface vertex pools as a clockwise winding of vertices suitable for direct rendering as a triangle fan primitive type. This means iwfSurfaces exported from GILES will usually have the **VERTICES_TRIFAN** flag set. Sometimes however, GILES will export faces which cannot be readily rendered as a triangle fan (perhaps because the face would contain degenerate triangles) and these faces are exported from GILES as indexed triangle lists.

#define VERTICES_TRISTRIP 0x4

This flags means that the vertices are ordered in such a way that the vertices could be rendered directly as a triangle strip primitive with a call to the DrawPrimitive function. Vertices are stored such that the first three vertices describe the first triangle of the surface, and every addition vertex creates a new triangle from that vertex and the previous two vertices in the list. Surfaces are never exported as trinagle strips from GILES.

#define VERTICES_TRILIST 0x8

This flag means that the vertices for the surface are arranged as a triangle list, where every three vertices in the list describe a triangle. This vertex pool could be directly rendered as a triangle list primitive type using the DrawPrimitive function. Triangle lists usually cause many duplicated vertices per surface and as such surfaces are never exported as triangle lists from GILES.

#define VERTICES_TRIFAN 0x10

This flag means that the vertices for this surface are stored in clockwise winding order suitable for rendering directly as a triangle fan primitive type. GILES exports most of its faces as triangle fans and only exports faces with the **VERTICES_INDEXED** flag when those faces cannot be described as a triangle fan. This saves file space by not requiring indices for surfaces that do not need them. If the loading application needs indices for these faces, it will have to generate them manually. See Chapter5Demo2 if you are not sure how to do this.

#define VERTICES_LINESTRIP 0x20

If this flag is set it means the vertices stored represent not triangles but rather a list of lines stored as a line strip. This vertex pool is in the correct order to be directly rendered by DirectX as a line strip primitive where the first two vertices define the first line segment and every addition vertex creates a new line segment from the current vertex and the previous one in the list. GILES never exports line strips so IWF files exported from GILES will never have this flag set.

#define VERTICES_LINELIST 0x40
This flag means that the vertex pool represents a list of lines where every pair of vertices in the pool defines a new line segment. This vertex pool is in the correct order to be rendered by DirectX as a line list primitive. GILES never exports line lists, so this flag will never be set for IWF vertex pools created in GILES.

#define VERTICES_POINTLIST 0x80
This flag means the vertex pool is intended to just be a pool of vertices with no connectivity order which are intended to be rendered as a series of points. Vertices in this order are suitable for rendering as a point list primitive in DirectX. GILES never exports point lists so this flag will not be set in IWF files created with GILES.

NOTE: After reviewing the above information we see that the surfaces that we will import from GILES will either be stored as triangle fans (in which case the vertex flag will be set to **VERTICES_TRIFAN**) or as triangle lists (in which case the **VERTICES_INDEXED** flag will be set).

USHORT VertexCount;
This variable holds the number of vertices that are in this surface's vertex pool.

UCHAR TexChannelCount;
This variable holds the number of texture coordinates that each vertex will contain.

UCHAR *TexCoordSize;
This array holds the dimensions of each texture coordinate set in the vertices used by this surface.

UCHAR IndexVPool;
This member can contain one of the following values defined in libIWF.h.

```
#define VPOOL_SURFACE           0x1
#define VPOOL_MESH              0x2
#define VPOOL_GLOBAL            0x3
```

Earlier we discussed that there can be a global vertex pool in the root of the file which all surfaces of all meshes index into, a per-mesh vertex pool which all the surfaces in a given mesh index into, or that each surface can store its own vertex pool. This member variable tells us which of these scenarios the current case is. GILES levels always store a vertex pool per face (each surface has its own list of vertices) so this member will always be set to **VPOOL_SURFACE** in IWF files created in GILES.

UCHAR IndexFlags;
Just as the VertexFlags member is used to describe how the vertices are stored, if indices exist for this face, this member tells us how those indices are stored. This allow us to know which primitive type to use when rendering this surface using DrawIndexedPrimitive. If a surface exported from GILES has indices, they will always be ordered to render the surfaces as an indexed triangle list. The following flags may be set (which are defined in libIWF.h):

#define INDICES_NONE 0x0
If this flag is set then no indices exist for this face.

#define INDICES_16BIT 0x1
If this bit is set it means that the indices stored are each 16 bits in size. GILES will always export 16 bit indices if indices exist for this surface.

#define INDICES_32BIT 0x2
If this bit is set it means the indices stored are each 32 bits in size.

#define INDICES_TRISTRIP 0x4
If this flag is set it means that indices exist and are arranged as an index triangle strip. The first three indices describe the first triangle and every additional index creates another triangle using the index and the two previous indices in the list. This flag will never be set with files created in GILES.

#define INDICES_TRILIST 0x8
This flag will be set when surfaces exported from GILES contain indices. The indices are ordered in such a way that they describe the face by indexing into the vertex pool as an indexed triangle list. Each set of three consecutive indices describes a separate triangle.

#define INDICES_TRIFAN 0x10
This flag means the surface has indices that are ordered as an indexed triangle fan. The first three indices describe the first triangle in the surface, and every additional index creates a triangle from that index, the previous index, and the first index in the list. This flag will never be set with files created in GILES.

#define INDICES_LINESTRIP 0x20
This flag means the indices describe a line strip of connected line segments where the first two indices define the first line and every additional index defines a new line from that vertex and the previous index in the list. This flag will never be set for IWF files created with GILES.

#define INDICES_LINELIST 0x40
If this bit is set it means the index list contains pairs of indices where each pair of indices defines a separate line segment. This flag will never be set in IWF files created with GILES.

#define INDICES_POINTLIST 0x80
This flag means that each index in the index list describes one vertex in the vertex pool which is to be rendered as a point primitive. This flag will never be set in IWF files created by GILES.

As you can see, for surfaces exported with GILES that contain indices, all will have the **`INDICES_TRILIST`** and the **`INDICES_16BIT`** flags set.

USHORT IndexCount;
This member describes how many indices are contained in the index list for this surface (if it exists).

iwfVertex *Vertices;
This is a pointer to the vertex data for this surface -- assuming the surface uses a surface vertex pool. This is always the case with GILES exported IWF files. Each vertex in the array is stored in a iwfVertex structure.

ULONG *Indices;
This is a pointer to an array of indices used by this surface -- assuming the surface uses indices. Each element in the array is stored as a 32 bit value. If the surface has the INDEXED_16BIT flag set then these values can be truncated and copied into 16 bit index buffers.

iwfVertex

The iwfVertex structure is flexible enough to deal with the many vertex types at our disposal (pre-transformed, lit, unlit, etc.). This means that there are some redundant members in this structure and we will copy only the vertex information we need into our vertex buffers.

GILES always exports vertices which have a world space position component (x, y, z) and a vertex normal. Texture coordinates may also be exported if the surface to which the vertex belongs is textured. The iwfVertex class also has many more members which are not used by GILES but may be useful for other development projects.

The iwfVertex class includes two constructors and two helper functions for allocating textures coordinates which are not shown here. If you are interested in looking at these member functions then please look in iwfObject.h. You will usually not want to instantiate iwfVertex objects unless you are writing an IWF file manually. In our case, we are simply interested in extracting the information from the instances created by the CFileIWF object that loaded the file.

```
class iwfVertex
{
   public:
       // Public Variables for This Class
        float    x;                    // Vertex Position X Component
        float    y;                    // Vertex Position Y Component
        float    z;                    // Vertex Position Z Component
        float    rhw;                  // Vertex position w component
        VECTOR3  Normal;               // Vertex Normal
        float    PointSize;            // Vertex point size
        ULONG    Diffuse;              // Diffuse vertex color
        ULONG    Specular;             // Specular vertex color
        UCHAR    TexChanCount;  // Number of tex coord channels
        float  **TexCoords;     // Texture coordinates.
};
```

While all of this may seem very complicated, using IWF files in our demos is actually very easy since we will simply extract the information we need from the CFileIWF object.

iwfMaterial

The CFileIWF::m_vpMaterialList is a variable of type **vectorIWFMaterial** which is define in iwfFile.h as :

```
typedef std::vector<iwfMaterial*>   vectorIWFMaterial;
```

The Material list vector contains a list of pointers to all of the materials used in the level. Each surface has a material index describing which material in this list should be used to render it. We will need to examine and extract the material information so that we can build an array of D3DMATERIAL9 structures in our application.

```
class iwfMaterial
{
 public:

    // Constructors & Destructors for This Class.
     iwfMaterial( );
    ~iwfMaterial( );

    // Public Variables for This Class
    char *Name;   // The character array containing the reference name.

    COLOUR_VALUE    Diffuse;       // Diffuse reflection component
    COLOUR_VALUE    Ambient;       // Ambient reflection component
    COLOUR_VALUE    Emissive;      // Emissive reflection component
    COLOUR_VALUE    Specular;      // Specular reflection component
    float           Power;         // Specular reflection power ratio
};
```

This structure is very similar to the **D3DMATERIAL9** structure. This makes copying data very easy. The iwfMaterial class also includes a Name member which can be assigned to a material in GILES. You may find this useful for identifying certain materials within the list by name.

The COLOUR_VALUE structure is a simple structure containing four floating point values and is analogous to the **D3DCOLORVALUE** structure used to store the red, green, blue and alpha components of a color. The COLOUR_VALUE structure is defined in libIWF.h.

```
typedef struct _COLOUR_VALUE    // Used to store material / light color values
{
    float       r;
    float       g;
    float       b;
    float       a;
} COLOUR_VALUE;
```

iwfEntity

We conclude our discussion with a brief look at the array of entities managed by the CFileIWF. The CFileIWF::m_vpEntityList is a variable of type **`vectorIWFEntity`** which is defined in iwfFile.h as shown:

```
typedef std::vector<iwfEntity*>      vectorIWFEntity;
```

In level editing terms, an entity can be described as anything that is not considered level geometry. An entity can contain the position and settings of a light source, the position in the world where the player should begin the level, a monster spawn point, the location of a particle emitter, or even a script reference linked to piece of geometry. The GILES Plug-In Example Kit included with this course provides you with a template for designing your own entity plug-ins for GILES that will be specific to your game engine.

In reality, an entity is simply a block of custom data that has been linked to your world. This entity data may or may not only be recognized by the developer of that entity. GILES does not care what data an entity holds, but all entities do have one characteristic: every one of them has a position and orientation in the world. Every entity object includes an EntityTypeID variable used by the loading application to identify what type of entity it is (a light , a monster spawn point, a player start point, etc.). A string label called 'Name' can be assigned so that different entities of the same type can be identified by name. A world matrix describing the position and the orientation of this entity in the world is also included. Finally, a custom data area that contains the actual entity information is included (which is completely different both in size and contents between entity types). The custom data area of a light entity contains all of the color, range, and attenuation values of the light. But a particle emitter entity for example might contain settings about how frequently a new particle should be created.

```
class iwfEntity : public IIWFObject
{
public:
     // Constructors & Destructors for This Class.
         iwfEntity( ULONG Size );
         iwfEntity( );
         virtual ~iwfEntity( );


     // Public Functions for This Class
          UCHAR   QueryType   ( ) { return OBJECT_TYPE_ENTITY; }
          bool    AllocateData( ULONG Size );


     // Public Variables for This Class
         USHORT   EntityTypeID;   // The Type Identifier for the entity.
         char    *Name;           // The entity reference name.
         MATRIX4  ObjectMatrix;   // The matrix assigned to the entity.
         ULONG    DataSize;       // The size of the entity's data area
         UCHAR   *DataArea;              // The entity's data area.
};
```

Unless you are using this class to create your own entities or write entities to an IWF file, you can ignore the member functions that aid in allocating memory for the entity. We are only interested in extracting the data from each entity that we want our demo application to support. In chapter 5, we will be searching the entity list for light entities with which we can build a list of D3DLIGHT9 structures describing all the lights in our scene.

Light Entities

To know whether an entity is a light entity we check the EntityTypeID member variable of the iwfEntity class. It will be set to the following value:

```
#define ENTITY_LIGHT              0x0010
```

If EntityTypeID equals 16 then we know that it is a light entity and that the matrix contains the position and orientation of the light and that the custom data area will contain all of the light properties. The D3DLIGHT9 structure requires point lights and spotlights to have a position so this can be extracted from the 4th row of the matrix. Directional lights and spotlights require a direction vector describing the direction of the light photons. This vector can be extracted from the 3rd row of the matrix.

The ***DataSize*** member describes how large the custom data area is in bytes. ***DataArea*** is a BYTE pointer to this information. To access the light information stored in this custom data area we need to know how the data is arranged. The iwfObject.h file defines a structure called **LIGHTENTITY** shown below. All our application will need to do is cast the ***DataArea*** pointer to a **LIGHTENTITY** pointer and we can extract all the information we require using its members. Most of the members of the **LIGHTENTITY** structure will be familiar to you since they are analogous to many of the members of the **D3DLIGHT9** structure:

```
struct LIGHTENTITY
{
        long         LightType;

        float        DiffuseRed;
        float        DiffuseGreen;
        float        DiffuseBlue;
        float        DiffuseAlpha;

        float        SpecularRed;
        float        SpecularGreen;
        float        SpecularBlue;
        float        SpecularAlpha;

        float        AmbientRed;
        float        AmbientGreen;
        float        AmbientBlue;
        float        AmbientAlpha;

        float       Range;
        float       FallOff;
        float       Attenuation0;
```

```
        float       Attenuation1;
        float       Attenuation2;

        float       Theta;
        float       Phi;
    };
```

The first member of this structure (LightType) can be set to one of the following values defined in LibIWF.h:

```
#define LIGHTTYPE_POINT                         0x0
#define LIGHTTYPE_SPOT                          0x1
#define LIGHTTYPE_DIRECTIONAL                   0x2
#define LIGHTTYPE_AMBIENT                       0x3
```

An ambient light entity is usually used to inform the application about the global ambient light setting the designer considers desirable for the level. The application can check for an ambient light entity and use it to set the global ambient light level of the scene using IDirect3DDevice9::SetRenderState.

The following example code extracts the 6th entity from the CFileIWF objects entity list, checks to see if it is a light entity, and if so, extracts the range information from the custom data area:

```
CFileIWF       File;
iwfEntity    * pEntity;
LIGHTENTITY  * pLightEntity;
float          LightRange;

// Load File
File.Load("Example.iwf");

//Extract 6th entity in list
pEntity = File.m_vpEntityList[5];

//If this is a light entity
if ( pEntity->EntityTypeID == ENTITY_LIGHT)
{
       // cast the custom data pointer to a LIGHTENTITY pointer
       plight = (LIGHTENTITY*) pEntity->DataArea;

       // Access range member in custom data area
       LightRange = pLight->Range;
}
```