Chapter 21

X Files

"Please explain to me the scientific nature of the whammy."
Special Agent Dana Scully, *The X-Files*, "Pusher"

21.1 Overview

In chapter 5, we saw how a scene is made up of a number of models and each model is constructed from primitives and vertices. The models are usually arranged into a hierarchy of coordinate frames related to each other by a transformation matrix.

While it is possible to construct scenes entirely from code, using so-called procedural models, the more common situation is to use a modeling tool to create scenes and scene elements. The modeler stores the information into a file and the application parses the file to construct the scene using device resources.

When it comes to file formats for storing scene data, you can invent your own or use an existing file format. Using an existing file format has the advantage that you can use existing libraries for reading and writing the format and stay focused on your application. You can leverage third party modelling and conversion tools with an existing published format.

The DirectX SDK uses the X file format for storing scene data. X files are an extensible, hierarchical file format that can directly represent the scene hierarchy and the associated vertices. Data is organized into a hierarchy of data nodes. The structure of each node is described by a template.

The SDK includes a collection of interfaces that allow an application to read and write X files containing scene data. Additional tools are provided to export scene data from modelling packages into X files. The specification for the format is published so you can use it in your own tools or even write your own parser if you desire.

First, we will describe the organizational structure of X files. The structure of an X file can be encoded as text or binary data. We will look at the text

Offset	Bytes	ASCII	Meaning
0	0x78, 0x6F, 0x66, 0x20	"xof "	X file signature
4	0x30, 0x33	"03"	major version number
6	0x30, 0x33	"03"	minor version number
8	0x74, 0x78, 0x74, 0x20	"txt "	text encoding
8	0x62, 0x69, 0x6E, 0x20	"bin "	binary encoding
8	0x74, 0x7A, 0x69, 0x70	"tzip"	compressed text encoding
8	0x62, 0x7A, 0x69, 0x70	"bzip"	compressed binary encoding
12	0x30, 0x30, 0x33, 0x32	"0032"	32-bit floating-point values
12	0x30, 0x30, 0x36, 0x34	"0064"	64-bit floating-point values

Table 21.1: X File Header

encoding first, followed by the binary encoding. With the file format out of the way, a summary of the templates provided by the SDK is given.

Next, we will look at the interfaces for reading and writing X files. After the interfaces have been covered, a general discussion of reading and writing X files will be given. Finally, we conclude the chapter with some simple X file examples and a sample program that demonstrates the writing a simple scene to an X file.

21.2 X File Structure

X files are organized into a hierarchy of data nodes, where the structure of each node is given by a template. The arrangement of the data nodes in the file can directly represent the hierarchy of the coordinate frames of a scene. All nodes in the file are described by a data template. The SDK includes a collection of predefined templates for use with the D3DX meshes. Applications can define their own templates if the existing templates don't suit the needs of the application.

X files support the concept of instancing, where references to previously defined nodes can be used instead of repeating the data. This is useful in graphics applications where a scene containing multiple copies of the same modeling element can be succinctly encoded by storing a single copy of the element and referencing the element multiple times.

The data in an X file may be encoded in a text or binary form. The organizational structure of the data remains the same in either case. Either encoding can be compressed for efficient storage. The text encoding is useful for development and experimentation, since the file can be created with any text editor.

21.2.1 X File Header

Every X file begins with a sequence of header bytes that identify the file as an X object file, the version of the file format, the encoding used for the file, and the

size of the floating-point values used in the file. Each of these items is specified as four ASCII characters. The structure of the header is shown in table 21.1.

21.2.2 Base Data Types

All data stored in X files must ultimately be decomposed into data in one of the following base data types:

```
byte   word   dword  string
char   sword   int    float
```

Unsigned integers of 8, 16 and 32 bits in size are represented by the `byte`, `word` and `dword` types. Signed integers of 8, 16 and 32 bits in size are represented by the `char`, `sword`, and `int` types. Floating-point values are represented by the `float` type. The size of the floating-point values, either 32-bit single precision or 64-bit double precision, is determined by the file header. ANSI strings are represented by the `string` type. Unicode strings are not supported in X files.

21.2.3 Templates

Following the file header, template definitions can appear in the file for a self-contained description of the data in the file. An application can also register templates with the file parser to avoid storing the templates in every data file. The parsing interfaces have no templates registered by default, requiring templates to be registered before any data can be read.

Templates organize data into a group, similar to a structure in C++. Each template consists of a name, a GUID, zero or more data members and an optional list of allowed nested templates. Each member has a data type and a name. The data type can be one of the base data types, the name of a defined template or an array. The type of the array elements can be a base type or a defined template. Each template definition lists the templates that can be nested within it. If a template definition omits the list of allowed templates, then no child templates may be nested within it and it is called "closed". A list of specific templates restricts the child nodes that may be nested within the template and is called a "restricted" template. A template may also indicate that any defined template may be nested within it and is called "open".

21.2.4 Data Nodes

Data nodes are instances of defined templates. A template must be defined before it can be instantiated as a node. Each node may include nested subnodes according to the node's template definition. Node instances can be shared through a reference to any defined node. Nodes may contain arbitrary binary data for unstructured information.

x-file ← *header data-list*
header ← `"xof "` *x-file-version file-encoding float-size*
x-file-version ← `"0303"`
file-encoding ← `"txt " | "bin " | "tzip" | "bzip"`
float-size ← `"0032" | "0064"`
data-list ← *template-or-node* | *template-or-node data-list*
template-or-node ← *template* | *data-node*

Figure 21.1: X file grammar

Symbol	Meaning
empty	Blank
identifier	Any valid identifier
guid-hex-string	GUID in string form
integer-value	A signed or unsigned integer constant
float-value	A floating-point value constant
string-value	An ANSI string in double quotes (")

Table 21.2: Meaning of terminal X file grammar tokens

21.2.5 Identifiers

Templates, data nodes and node members can have names. Each name is given as a valid identifier. Identifiers must start with an alphabetic character, followed by zero or more alphabetic, numeric or underscore characters. This is similar to the identifier rules for C++ except that the X file identifiers cannot begin with an underscore. Reserved words in the text encoding are not valid identifiers. The parsing interfaces provided by the SDK compare identifiers in a case insensitive fashion. However, you should still use consistent case in your identifiers between the code and the template definitions. Other parsing libraries may not treat identifiers as case insensitive.

21.3 Text Encoding

The text encoding of X files allows them to be easily created in an ordinary text editor. The text encoding also allows external scripts or programs without access to the SDK libraries to generate data files for a Direct3D application.

Once the header has been read from the file, the remainder of a file in the text encoding is parsed as a sequence of tokens. The tokens consist of a number of reserved words and punctuation tokens. Whitespace characters in the text encoding, except for those inside a string value, are ignored and are used only to separate tokens. Punctuation tokens do not require separating whitespace as the punctuation characters themselves serve as the separator. The reserved words in the text encoding are not case sensitive; "`dword`" is treated identically to "`DWORD`".

template ← `"template"` *identifier* `"{"` *template-body* `"}"`
template-body ← *guid member-list restrictions*
guid ← `"<"` *guid-hex-string* `">"`
member-list ← *empty* | *member member-list*
member ← *type identifer* `";"`
type ← *base-type* | *array-type* | *identifier*
base-type ← `"byte"` | `"char"` | `"word"` | `"sword"`
| `"dword"` | `"int"` | `"string"` | `"float"`
array-type ← `"array"` *base-type dimension-list*
dimension-list ← *dimension* | *dimension dimension-list*
dimension ← `"["` *dimension-size* `"]"`
dimension-size ← *integer* | *identifier*
restrictions ← *empty* | `"[...]"` | `"["` *restriction-list* `"]"`
restriction-list ← *restriction* | *restriction* `","` *restriction-list*
restriction ← `"binary"` | *identifier* | *identifier guid*

Figure 21.2: Template grammar

The grammar for the text encoding of an X file is given in figure 21.1. In the grammar figures given for the text encoding of X files, literal text is shown in a fixed-width font and surrounded by quotation marks. Grammar symbols are shown in an italic font and the expansions for each non-terminal symbol show alternative expansions separated by a vertical bar. The terminal symbols shown in table 21.2 are not defined in the grammar explicitly.

The text encoding allows comments to appear within the X file. Comments begin with either a double-slash character sequence (`//`) or a hash character (`#`). In both cases, the comment starts with the opening character sequence and continues to the end of the line. The comments may appear anywhere on a line.

21.3.1 Reserved Words

The following words are reserved in the text encoding. These words should not be used for identifiers.

`array`	`binary`	`binary_resource`	`char`
`cstring`	`double`	`dword`	`float`
`lpstr`	`sdword`	`string`	`sword`
`template`	`uchar`	`ulonglong`	`unicode`
`void`	`word`		

21.3.2 Templates

The grammar for a template definition is given in figure 21.2. Every template has an identifier and a GUID. The GUID will be exposed to your code through the parsing interfaces, but not the identifier, so you should think of the GUID as the real identifier for a template. The GUID for a template must be unique.

Always generate new GUIDs when you change or modify any template that has been published. This use of GUIDs is similar to their use in COM for providing unique identifiers for immutable interfaces. You should consider any published template definition to be immutable; if you need to change the data stored inside the template, publish a new template that replaces or augments the existing template. Visual C++ includes the `GUIDGEN` tool for generating new GUIDs to be pasted into an X file.

Closed templates must have at least one data member. Restricted or open templates can have zero data members, acting as a container for other nodes only:

```
template Stuffing
{
    <61daec0a-b6e9-4a4f-96e1-ee63dec5fb0b>
    [...]
}
```

The data members within a template are given similarly to the declaration of members of a C++ structure. Each member is terminated with a semi-colon. Array members are preceded by the keyword `array`, followed by the base type of the array, the name of the member and a dimension list. X file templates can describe variable length arrays as well as fixed-length arrays. For a variable-length array, the size of the array must be defined as a member of the template before the array member. The dimension of the array is given as the name of the member containing the size of the array. See section 21.5 for examples of template definitions in the text encoding.

21.3.3 Data Nodes

The grammar for the data in a node is given in figure 21.3. Each node begins by naming the template defining the structure of the node. Next, an optional identifier may be given to this instance of the template. Curly brace characters surround the body of the node data. All data, including nested data, between the open brace and the close brace is part of the data node. The node body consists of an optional GUID that identifies this particular node instance, the data for the members of the node and any nested nodes contained within this node.

For nodes with data members, the data for each member is given in the order they are listed in the template, followed by a semi-colon. For array members, the data for each array element is given and separated from other array elements by a comma. Note that array elements are separated by commas, while data members are terminated by semi-colons.

The separating commas and terminating semi-colons also apply to any members whose type is a defined template, as the following example illustrates. In this example, the `Point3` template defines a three-dimensional point with members `x`, `y` and `z`. The `PolyLine3` template defines a polyline as an array of

data-node ← *identifier optional-identifier* `"{"` *optional-guid*
member-data-list nested-data-list `"}"`
optional-identifier ← *empty* | *identifier*
optional-guid ← *empty* | *guid*
member-data-list ← *member-data* | *member-data member-data-list*
member-data ← *data-value* `";"`
data-value ← *integer-value* | *float-value*
| *string-value* | *member-data-list*
| *array-value*
array-value ← *integer-list* | *float-list*
| *string-list* | *node-list*
integer-list ← *integer* | *integer-value* `","` *integer-list*
float-list ← *float* | *float-value* `","` *float-list*
string-list ← *string* | *string-value* `","` *string-list*
node-list ← *member-data-list* | *data-node* `","` *node-list*
nested-data-list ← *empty* | *nested-data nested-data-list*
nested-data ← *data-node* | *data-reference*
data-reference ← `"{"` *node-reference* `"}"`
node-reference ← *identifier* | *guid* | *identifier guid*

Figure 21.3: Data node grammar

`Point3` templates. Finally, an instance of the `PolyLine3` template provides the data for a polyline with two segments and therefore three points.

```
template Point3
{
    <fc53dae9-97c0-4d03-b38c-00e9b5467675>
    float x;
    float y;
    float z;
}

template PolyLine3
{
    <1f822819-e01c-481d-8a8e-3a1f1bebf47f>
    dword numPoints;
    Point3 points[numPoints];
}

PolyLine3
{
    3;
    0;0;0;,
    1;0;0;,
    2;0;0;;
```

```
}
```

21.3.4 Nested Data

There are three kinds of nested data that may appear within a node: binary data, data nodes and data references. Nested data nodes have the same structure as top-level data nodes.

Binary data may be included in an X file with a text encoding, but the data itself will be written in binary form. The X file format provides a special token to switch between the text and binary encodings in the middle of the file. The interfaces provided in the SDK handle these details of including binary objects.

References to data nodes defined earlier in the file can be included as children of a data node. Data nodes may be referenced by node's identifier or its GUID. A reference is enclosed in curly brace characters. A reference may be by name or by GUID. The name refers to the optional identifier associated with a previously defined node. The GUID refers to the optional GUID associated with a previously defined node. Be careful not to confuse these identifiers and GUIDs with template identifiers and the template GUID.

The following example demonstrates how an `Animation` template references a `Frame` template defined earlier in the file. The required member data for the `Frame` and `AnimationKey` templates is omitted for clarity. The first `Animation` node references the `Scene_Root` node by name, while the second `Animation` node references the `Child` node by name and GUID.

```
Frame Scene_Root
{
    // ... frame data ...

    Frame Child
    {
        <cedb2817-fc86-44fb-b78f-cc015798ac49>
        // ... frame data ...
    }
}

AnimationSet
{
    Animation
    {
        AnimationKey
        {
            // ... keyframe data ...
        }
        { Scene_Root }
    }
```

Token	Value	Token	Value
TOKEN_ARRAY	52	TOKEN_CANGLE	17
TOKEN_CBRACE	11	TOKEN_CBRACKET	15
TOKEN_CHAR	44	TOKEN_COMMA	19
TOKEN_CPAREN	13	TOKEN_CSTRING	51
TOKEN_DOT	18	TOKEN_DOUBLE	43
TOKEN_DWORD	41	TOKEN_FLOAT	42
TOKEN_LPSTR	49	TOKEN_OANGLE	16
TOKEN_OBRACE	10	TOKEN_OBRACKET	14
TOKEN_OPAREN	12	TOKEN_SEMICOLON	20
TOKEN_SDWORD	47	TOKEN_SWORD	46
TOKEN_TEMPLATE	31	TOKEN_UCHAR	45
TOKEN_UNICODE	50	TOKEN_VOID	48
TOKEN_WORD	40		

Table 21.3: X file tokens

```
    Animation
    {
        AnimationKey
        {
            // ... keyframe data ...
        }
        { Child <cedb2817-fc86-44fb-b78f-cc015798ac49> }
    }
}
```

21.4 Binary Encoding

In the binary encoding for an X file, all data is represented as a sequence of tokens. Each token is identified by a `WORD` sized value, followed by any additional data for the token. The tokens in table 21.3 are stored only by their token values, while the tokens in table 21.4 are followed by a data record giving the additional information. All data in the binary encoding, including token `WORD`s, is stored in little-endian format.

The simple tokens generally correspond directly to a reserved word token or a punctuation character token. The pairs of matching opening and closing punctuation (`<>`, `{}`, `[]`, `()`) are each indicated by an opening token and a closing token. For instance, the opening and closing parenthesis tokens are represented by `TOKEN_OPAREN` and `TOKEN_CPAREN`.

In the binary encoding, data is aggregated into the largest possible unit. Returning to the polyline example from page 727, the text encoding separates different members within the `Point3` values in the `PolyLine3` template instance, while the binary encoding would lump all the values into a single list of nine `float`s.

Token	Value	Token	Value
TOKEN_FLOAT_LIST	7	TOKEN_GUID	5
TOKEN_INTEGER	3	TOKEN_INTEGER_LIST	6
TOKEN_NAME	1	TOKEN_STRING	2

Table 21.4: X file record tokens

21.4.1 Token Records

Token records contain the data associated with elements of the X file structure such as identifiers, array dimensions and member names within a template definition as well as the values within a data node. The following tables describe the structure of the token records.

GUID

Field	Type	Size	Contents
token	WORD	2	TOKEN_GUID
data1	DWORD	4	GUID data field 1
data2	WORD	2	GUID data field 2
data3	WORD	2	GUID data field 3
data4	BYTE[8]	8	GUID data field 4

Integer

Field	Type	Size	Contents
token	WORD	2	TOKEN_INTEGER
value	DWORD	4	integer value

Integer List

Field	Type	Size	Contents
token	WORD	2	TOKEN_INTEGER_LIST
count	DWORD	4	list length
list	DWORD[count]	4×count	integer list

Float List

Field	Type	Size	Contents
token	WORD	2	TOKEN_FLOAT_LIST
count	DWORD	4	list length
list	float[count]	4×count	float list

Name

Field	Type	Size	Contents
token	WORD	2	TOKEN_NAME
count	DWORD	4	length of name in bytes
name	BYTE[count]	count	ASCII name

String

Field	Type	Size	Contents
token	WORD	2	TOKEN_STRING
count	DWORD	4	length of string in bytes
string	BYTE[count]	count	ASCII string
terminator	DWORD	4	TOKEN_SEMICOLON or TOKEN_COMMA

21.5 Predefined Templates

The SDK includes a number of predefined templates for use with the D3DX functions that load and save ID3DXMesh objects. These templates originate with so-called "retained mode" graphics from earlier releases of DirectX. Some of the templates in this section are deprecated or for the internal use of D3DX and are described only for completeness. An application should create custom templates when there is no predefined template that exactly matches the semantics intended by the application.

The predefined templates can be grouped into several categories: mesh templates, progressive mesh templates, skinned mesh templates, animation templates and miscellaneous templates. The mesh templates store data relating to ID3DXMesh objects representing indexed triangle lists. The progressive and skinned mesh templates define information specific to those mesh representations. The animation templates define keyframe data for simple mesh animations. The miscellaneous category picks up any remaining templates.

The SDK includes several header files for use with the predefined templates. The <rmxfguid.h> header file uses the DEFINE_GUID macro to declare identifiers for the template GUIDs. The DEFINE_GUID macro will define GUID instances as data if INITGUID is defined when the Windows header <objbase.h> is included. Otherwise, DEFINE_GUID will declare external references to the GUIDs.

Most of the GUIDs used by the SDK are defined in the library dxguid.lib, so it is not necessary to use INITGUID; simply link against the library to define the GUIDs for use with your application. This method is preferred over using INITGUID, which can cause multiply defined symbols if not used properly.

In addition to the template GUIDs, the SDK contains the header file <rmxftmpl.h>. This header contains template definitions for most of the predefined templates. This header declares an array of unsigned chars to hold the template definition. Since this header defines data, you should include it in at most one source file in your project. Never include it through another header file, or multiply defined symbol errors will occur.

The file `xskinexptemplates.h` is included in the `Extras` subdirectory of the SDK distribution with the source for the X file export plugins for various modellers. It defines the templates `FVFData`, `Patch`, `PatchMesh`, `PMAttributeRange`, `PMInfo`, `PMVSplitRecord`, `SkinWeights`, `VertexDuplicationIndices` and `XSkinMeshHeader`. These templates are registered by the D3DX functions that load skinned meshes.

In the descriptions that follow, each template is give by its text encoding. Its required member data is described and references are given to any related templates given in this section.

21.5.1 Animation

The `Animation` template specifies a set of key frame animation values to be applied to a coordinate frame in the hierarchy. There are no required data members in the template. The frame to be animated is given as a reference node contained in the `Animation` ndoe. The key frame information is contained as one or more `AnimationKey` nodes. An `AnimationOptions` node may also be included to specify options for the animation. The template is open, allowing any application-specific templates may be included as child nodes.

```
template Animation
{
    <3D82AB4F-62DA-11CF-AB39-0020AF71E433>
    [...]
}
```

Related Templates

See the `AnimationKey` template on page 732, the `AnimationOptions` template on page 733, the `AnimationSet` template on page 734, the `Frame` template on page 737, the `TimedFloatKeys` template on page 751, and the `FloatKeys` template on page 736.

21.5.2 Animation Key

The `AnimationKey` template contains an array of key frame data for an interpolating animation of a coordinate frame's transformation matrix. The `keyType` member specifies the type of key framing interpolation to apply to the coordinate frame. The values and their meaning are summarized in table 21.5.

```
template AnimationKey
{
    <10DD46A8-775B-11CF-8F52-0040333594A3>
    DWORD keyType;
    DWORD nKeys;
    array TimedFloatKeys keys[nKeys];
}
```

Key Type	Meaning	Number of Values	Data Type
0	rotation	4	quaternion
1	scaling	3	vector
2	translation	3	vector
4	matrix	16	matrix

Table 21.5: Animation key types

For a rotation key, the key frame data consists of four `float`s for each time value. The values represent a rotation expressed as a quaternion, with the components of the quaternion stored in the order w, x, y, and z. Note that the memory layout of a `D3DXQUATERNION` gives the components in the order x, y, z, and w.

For a scaling key, the key frame data consists of three `float`s for each time value. The values represent a scaling vector stored in the order s_x, s_y, s_z.

For a translation key, the key frame data consists of three `float`s for each time value. The values represent a translation vector stored in the order t_x, t_y, t_z.

For a matrix key, they key frame data consists of 16 `float`s for each time value. The values represent an arbitrary 4×4 matrix stored in the same order as the memory layout of `D3DXMATRIX` and `D3DMATRIX`, a_{11}, a_{12}, a_{13}, a_{14}, a_{21}, $\ldots$, a_{44}.

Each `AnimationKey` template contains an array of `TimedFloatKeys` in the `keys` member and the size of the array is given by the `nKeys` member. As all the keys have the same type, each element of the array will have the same number of `float`s defining the key.

Related Templates

See the `Animation` template on page 732, the `AnimationSet` template on page 734, the `TimedFloatKeys` template on page 751, and the `FloatKeys` template on page 736.

21.5.3 Animation Options

The `AnimationOptions` template gives two options that affect the entire animation. The `openclosed` member indicates whether the animation should repeat in an endless loop (closed), or whether it should play through once (open). The value zero indicates a closed animation and the value one indicates an open animation. The `positionquality` member describes the interpolation quality used for translation key frame data. A value of zero indicates the interpolating splines should be used, while a value of one indicates that linear interpolation should be used.

```
template AnimationOptions
```

```
{
    <E2BF56C0-840F-11CF-8F52-0040333594A3>
    DWORD openclosed;
    DWORD positionquality;
}
```

Related Templates

See the `Animation` template on page 732.

21.5.4 Animation Set

A group of `Animation` templates may be combined with the `AnimationSet` template. This allows multiple kinds of key frame data to be associated with a single timeline, with each type of key contained in a single `Animation` node. It also allows multiple animated movements on the same model to appear in a single file without duplicating the model. For instance, one animation set might contain a looping walking sequence, while another contains a non-looping jump sequence.

```
template AnimationSet
{
    <3D82AB50-62DA-11CF-AB39-0020AF71E433>
    [Animation <3D82AB4F-62DA-11CF-AB39-0020AF71E433>]
}
```

Related Templates

See the `Animation` template on page 732.

21.5.5 Boolean

This template defines a simple boolean value. The `truefalse` member has the value zero to indicate false and the value one to indicate true.

```
template Boolean
{
    <537DA6A0-CA37-11D0-941C-0080C80CFA7B>
    DWORD truefalse;
}
```

Related Templates

See the `Boolean2d` template on page 735 and the `MeshFaceWraps` template on page 743.

21.5.6 Boolean 2D

This template defines two booleans indicating a truth value in each of the u and v directions. It is used to define the texture coordinate wrapping topology for faces in a mesh.

```
template Boolean2d
{
    <4885AE63-78E8-11CF-8F52-0040333594A3>
    Boolean u;
    Boolean v;
}
```

Related Templates

See the `Boolean` template on page 734 and the `MeshFaceWraps` template on page 743.

21.5.7 Color RGB

This template defines an RGB color as a triplet of three `float`s. This template is used for the specular and emissive material colors of meshes.

```
template ColorRGB
{
    <D3E16E81-7835-11CF-8F52-0040333594A3>
    FLOAT red;
    FLOAT green;
    FLOAT blue;
}
```

Related Templates

See the `Material` template on page 740.

21.5.8 Color RGBA

This template defines an RGBA color as four `float` values. This template is used for the diffuse color of meshes.

```
template ColorRGBA
{
    <35FF44E0-6C7C-11CF-8F52-0040333594A3>
    FLOAT red;
    FLOAT green;
    FLOAT blue;
    FLOAT alpha;
}
```

Related Templates

See the `IndexedColor` template on page 739 and the `Material` template on page 740.

21.5.9 Coords 2D

This template defines two-dimensional texture coordinates as two `float` values. It is used for two-dimensional texture coordinates of meshes.

```
template Coords2d
{
    <F6F23F44-7686-11CF-8F52-0040333594A3>
    FLOAT u;
    FLOAT v;
}
```

Related Templates

See the `MeshTextureCoords` template on page 744.

21.5.10 External Visual

This is a legacy template whose contents and usage are undefined. An application should not use this template.

```
template ExternalVisual
{
    <98116AA0-BDBA-11D1-82C0-00A0C9697271>
    Guid guidExternalVisual;
    [...]
}
```

Related Templates

See the `Guid` template on page 738.

21.5.11 Float Keys

This template defines an array of `float`s that define a single key value for use in a key frame animation. The `values` member holds the array and the `nValues` member gives the size of the array.

```
template FloatKeys
{
    <10DD46A9-775B-11CF-8F52-0040333594A3>
    DWORD nValues;
    array FLOAT values[nValues];
}
```

Related Templates

See the `AnimationKey` template on page 732, and the `TimedFloatKeys` template on page 751.

21.5.12 Frame

The frame template is used to indicate a coordinate frame. The template is open so that application-specific data can be associated with each defined coordinate frame. Usually a frame contains a `FrameTransformMatrix` node giving the matrix defining this coordinate frame relative to its parent frame and one or more `Mesh` nodes giving meshes defined in the coordinate frame. Coordinate frames relative to this node are given as child `Frame` nodes.

```
template Frame
{
    <3D82AB46-62DA-11CF-AB39-0020AF71E433>
    [...]
}
```

Related Templates

See the `FrameTransformMatrix` template on page 737, and the `Mesh` template on page 741.

21.5.13 Frame Transform Matrix

The `FrameTransformMatrix` template defines a coordinate transformation for a `Frame` node relative to the parent node of the `Frame`. A `Frame` node should contain at most a single `FrameTransformMatrix` node. The `frameMatrix` member contains the transformation matrix that maps coordinates from the coordinate system of the parent node to the coordinate system of the child node.

```
template FrameTransformMatrix
{
    <F6F23F41-7686-11CF-8F52-0040333594A3>
    Matrix4x4 frameMatrix;
}
```

Related Templates

See the `Frame` template on page 737 and the `Matrix4x4` template on page 741.

21.5.14 FVF Data

The predefined mesh-related templates do not define templates for all the vertex data that can be stored in a mesh. For instance, the predefined templates do not allow for three-dimensional texture coordinates or multiple texture coordinate

sets per vertex. The FVFData template is used as a container for additional per-vertex data specified by an FVF code. This node will be used by the SDK modeller exporters and D3DX mesh saving functions when the mesh has data not supported by the other predefined mesh templates. It will appear in the hierarchy as a child node of the corresponding Mesh node.

The dwFVF member contains the FVF code of the data stored in the data array member. The history of this template is such that the dwFVF member will always have the D3DFVF_XYZ bit set, but the data array will not contain position information. To get the actual FVF code of the associated data, mask off the position bits with D3DFVF_POSITION_MASK. If you write this template in your own applications, you should not set any position bits.

The size of the data array member is given by the nDWords member and will be the product of the number of vertices in the mesh and the size of the vertex components corresponding to the FVF code. For instance, if the associated FVF code corresponds to a three-dimensional texture coordinate set and a specular color for a mesh with 45 vertices, then the nDWords member would have a value of 180. Three DWORDs for the texture coordinate set and one DWORD for the specular color, for a total of four DWORDs per vertex, with 45 vertices.

The data array member contains the additional FVF data as raw DWORDs. If the associated vertex component data consists of floats, such as vertex blend weights or texture coordinates, then the data must be reinterpreted as floats.

```
template FVFData
{
    <B6E70A0E-8EF9-4E83-94AD-ECC8B0C04897>
    DWORD dwFVF;
    DWORD nDWords;
    array DWORD data[nDWords];
}
```

Related Templates

See the Mesh template on page 741.

21.5.15 Guid

This template defines a globally unique identifier (GUID). While it was created for use with the ExternalVisual template, it can be used whenever an application needs to store a GUID in a template.

```
template Guid
{
    <A42790E0-7810-11CF-8F52-0040333594A3>
    DWORD data1;
    WORD data2;
    WORD data3;
    array UCHAR data4[8];
```

```
}
```

Related Templates

See the `ExternalVisual` template on page 736.

21.5.16 Header

The `Header` template provides the retained-mode file version information. The `major` and `minor` members give the version number and the `flags` member specifies a collection of bit flags. When used to identify the retained-mode format of the file, the values used should be one for `major`, zero for `minor` and one for `flags`. If you wish to provide your own version information specific to your application, you should create a custom template to avoid confusion with retained-mode file versions.

```
template Header
{
    <3D82AB43-62DA-11CF-AB39-0020AF71E433>
    WORD major;
    WORD minor;
    DWORD flags;
}
```

21.5.17 Indexed Color

The `IndexedColor` template is used to associate a diffuse color with a vertex in a mesh. The `IndexedColor` template is used as a data member of the `MeshVertexColors` template. The `index` member gives the zero-based index of the vertex associated with the color. The index is relative to the array of vertices given in the `Mesh` node containing the `MeshVertexColors` node. The `indexColor` member gives the RGBA diffuse color associated with the vertex as four `float`s.

```
template IndexedColor
{
    <1630B820-7842-11CF-8F52-0040333594A3>
    DWORD index;
    ColorRGBA indexColor;
}
```

Related Templates

See the `ColorRGBA` template on page 735 and the `MeshVertexColors` template on page 745.

21.5.18 Inline Data

The `InlineData` template is used only by the `ProgressiveMesh` template to define the data describing a progressive mesh. This template is no longer actively used and is only described here for completeness. The data is given as a binary node when the template is instantiated.

```
template InlineData
{
    <3A23EEA0-94B1-11D0-AB39-0020AF71E433>
    [BINARY]
}
```

Related Templates

See the `ProgressiveMesh` template on page 748.

21.5.19 Material

The `Material` template describes the material properties of a mesh. It appears as a child node of a `MeshMaterialList` node, which in turn is a child of a `Mesh` node. The template is open, so that any additional material parameters needed by the application can be added as a child node of the `Material` node. In particular, you will notice that the template does not describe all the data that can be specified with the `D3DMATERIAL9` structure—the emissive color has no alpha component and the ambient color is not stored in this template. Textured meshes will use the `TextureFilename` node as a child of the `Material` node to give the filename of the texture associated with the mesh.

The `faceColor` member gives the diffuse color material property and corresponds to the `Diffuse` member of the `D3DMATERIAL9` structure. The `power`, `specularColor` and `emissiveColor` members correspond to the `Power`, `Specular` and `Emissive` members of the `D3DMATERIAL9` structure. When initializing a `D3DMATERIAL9` structure from the data in the `Material` template, be sure to set the alpha component of the `Specular` and `Emissive` members and the `Ambient` member to sensible values.

```
template Material
{
    <3D82AB4D-62DA-11CF-AB39-0020AF71E433>
    ColorRGBA faceColor;
    FLOAT power;
    ColorRGB specularColor;
    ColorRGB emissiveColor;
    [...]
}
```

Related Templates

See the `ColorRGB` template on page 735, the `MeshMaterialList` template on page 743, and the `TextureFilename` template on page 750.

21.5.20 Material Wrap

The `MaterialWrap` template is deprecated and should not be used. It is shown here for completeness.

```
template MaterialWrap
{
    <4885AE60-78E8-11CF-8F52-0040333594A3>
    Boolean u;
    Boolean v;
}
```

Related Templates

See the `Boolean` template on page 734.

21.5.21 Matrix 4x4

The `Matrix4x4` template defines a 4×4 homogeneous transformation matrix. It is used as a member in the `FrameTransformMatrix` and `SkinWeights` templates. You can use this template whenever you need to store additional homogeneous coordinate transformation matrices in your X files. The `matrix` member gives the transformation matrix as an array of 16 `float`s.

```
template Matrix4x4
{
    <F6F23F45-7686-11CF-8F52-0040333594A3>
    array FLOAT matrix[16];
}
```

Related Templates

See the `FrameTransformMatrix` template on page 737 and the `SkinWeights` template on page 749.

21.5.22 Mesh

The `Mesh` template is the main template used by most X files. The required members of this template store an array of `vertices` and an array of `faces`. D3DX uses the `Mesh` and related templates to read and write indexed triangle lists.

While the mesh template allows for each face to have an arbitrary number of vertices, you will get the best results if you restrict each face to a triangle

when using D3DX and the X file parsing interfaces for meshes. If you have complex polygons in your internal data structures, you should consider using a custom template and parsing that template yourself, or tessellating those complex polygons into triangles.

```
template Mesh
{
    <3D82AB44-62DA-11CF-AB39-0020AF71E433>
    DWORD nVertices;
    array Vector vertices[nVertices];
    DWORD nFaces;
    array MeshFace faces[nFaces];
    [...]
}
```

The `Mesh` template is open, allowing you to associate arbitrary application specific data with a mesh. A child `MeshMaterialList` node can be used to associate material properties with the triangles in the mesh. A `MeshNormals` node will provide normals for each of the vertices in the `Mesh`. The topology of the faces described in the `MeshNormals` node must match the topology of the faces described in the parent `Mesh` node. A `MeshTextureCoords` child node supplies a single set of two-dimensional texture coordinates for the faces described in the parent `Mesh`. To provide texture coordinates with other dimensionality, or additional texture coordinate sets, use a `FVFData` child node. A `MeshVertexColors` child node supplies diffuse colors for the vertices. The number of colors provided should match the number of vertices defined in the parent `Mesh` node. A `RightHanded` node indicates that whether the mesh is defined in a left-handed or right-handed coordinate system. Without this node, a general purpose X file application must guess the handedness of a `Mesh` node or allow a user to flip the handedness interactively. The `VertexDuplicationIndices` node will be written by D3DX when adjacency information is given to one of the mesh saving routines.

Related Templates

See the `FVFData` template on page 737, the `MeshFace` template on page 742, the `MeshFaceWraps` template on page 743, the `MeshMaterialList` template on page 743, the `MeshNormals` template on page 744, the `MeshTextureCoords` template on page 744, the `MeshVertexColors` template on page 745, the `RightHanded` template on page 749, the `Vector` template on page 751 and the `VertexDuplicationIndices` template on page 752.

21.5.23 Mesh Face

The `MeshFace` template defines a polygonal face as an array of vertex indices. Although the template allows each face to have a different number of vertices,

the best operation will be obtained when all faces are triangles. The `faceVertexIndices` array member gives the zero-based indices into the vertex array of the parent `Mesh` node. The winding order of the triangles is defined by the order in which the indices are given in the array.

```
template MeshFace
{
    <3D82AB5F-62DA-11CF-AB39-0020AF71E433>
    DWORD nFaceVertexIndices;
    array DWORD faceVertexIndices[nFaceVertexIndices];
}
```

Related Templates

See the `Mesh` template on page 741.

21.5.24 Mesh Face Wraps

The `MeshFaceWraps` template gives the two-dimensional texture coordinate wrapping topology for each face. The `faceWrapValues` array member should contain one element for each face in the parent `Mesh` node.

```
template MeshFaceWraps
{
    <ED1EC5C0-C0A8-11D0-941C-0080C80CFA7B>
    DWORD nFaceWrapValues;
    array Boolean2d faceWrapValues[nFaceWrapValues];
}
```

Related Templates

See the `Boolean2d` template on page 735 and the `Mesh` template on page 741.

21.5.25 Mesh Material List

The `MeshMaterialList` template gives the material property definitions for the faces in a parent `Mesh` node. The `nFaceIndexes`[1] member should be equal to the number of faces in the parent `Mesh` node. For each face, a zero-based material index is given in the array.

The material properties are not provided as data members in the `MeshMaterialList` template. Rather, the materials are given as child nodes of the `MeshMaterialList` node. As child nodes, the materials can be listed explicitly, or can be listed by reference allowing the same material to be shared among several meshes in a single file. The materials are numbered in the order they appear as child nodes, beginning with the number zero. The `nMaterials` member gives the number of child material nodes in the `MeshMaterialList` node.

[1] *sic* – the plural of index is indices, not indexes.

```
template MeshMaterialList
{
    <F6F23F42-7686-11CF-8F52-0040333594A3>
    DWORD nMaterials;
    DWORD nFaceIndexes;
    array DWORD faceIndexes[nFaceIndexes];
    [Material <3D82AB4D-62DA-11CF-AB39-0020AF71E433>]
}
```

Related Templates

See the `Material` template on page 740 and the `Mesh` template on page 741.

21.5.26 Mesh Normals

The `MeshNormals` template gives per-vertex normals for each of the faces defined in the parent `Mesh` node. The normals are specified for each vertex in the mesh by a zero-based index into the array of `Vector`s in the `normals` member. The number of faces and the number of vertices in each face must match the corresponding amounts in the parent `Mesh` node.

```
template MeshNormals
{
    <F6F23F43-7686-11CF-8F52-0040333594A3>
    DWORD nNormals;
    array Vector normals[nNormals];
    DWORD nFaceNormals;
    array MeshFace faceNormals[nFaceNormals];
}
```

Related Templates

See the `Mesh` template on page 741, the `MeshFace` template on page 742 and the `Vector` template on page 751.

21.5.27 Mesh Texture Coords

The `MeshTextureCoords` template gives a single set of two-dimensional texture coordinates for each of the vertices in the parent `Mesh` node. If multiple sets of texture coordinates are needed, or texture coordinates whose dimensionality is other than two, then the `FVFData` template should be used. The `nTextureCoords` member should equal the number of vertices in the parent `Mesh` node.

```
template MeshTextureCoords
{
    <F6F23F40-7686-11CF-8F52-0040333594A3>
    DWORD nTextureCoords;
```

```
    array Coords2d textureCoords[nTextureCoords];
}
```

Related Templates

See the `Coords2d` template on page 736 and the `Mesh` template on page 741.

21.5.28 Mesh Vertex Colors

The `MeshVertexColors` template gives the diffuse color for the vertices in a parent `Mesh` node. The `nVertexColors` member should equal the number of vertices in the parent `Mesh` node. The `vertexColors` member gives the color of each vertex with an index into the vertex array of the parent `Mesh` node and an RGBA color.

```
template MeshVertexColors
{
    <1630B821-7842-11CF-8F52-0040333594A3>
    DWORD nVertexColors;
    array IndexedColor vertexColors[nVertexColors];
}
```

Related Templates

See the `IndexedColor` template on page 739 and the `Mesh` template on page 741.

21.5.29 Patch

The `Patch` template gives a Bézier surface patch as an array of indices into an array of control points. The `nControlIndices` member must be either 10, 15, or 16 for a cubic Bézier triangular patch, a quartic Bézier triangular patch, or a cubic Bézier rectangular patch, respectively. The order in which the indices are given corresponds to the topology of the control points within the patch and is shown in figure 21.4.

```
template Patch
{
    <A3EB5D44-FC22-429D-9AFB-3221CB9719A6>
    DWORD nControlIndices;
    array DWORD controlIndices[nControlIndices];
}
```

Related Templates

See the `PatchMesh` template on page 747.

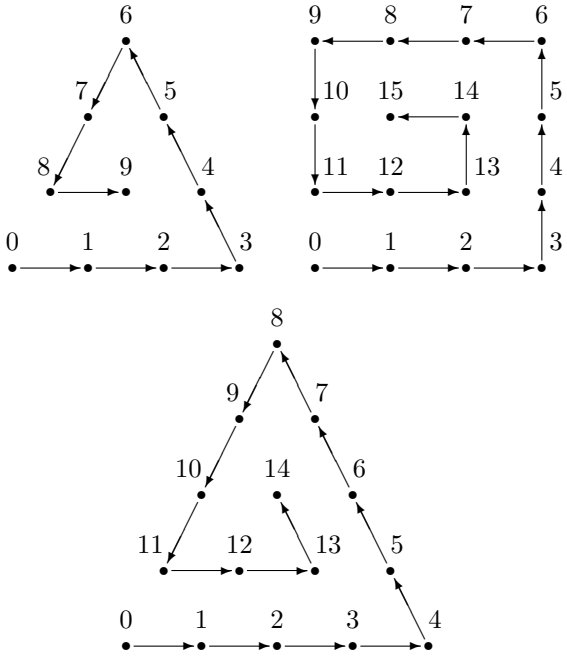

Figure 21.4: Order of indices for Bézier patches. The upper left shows the indices for a cubic Bézier triangular patch. The upper right shows the indices for a cubic Bézier rectangular patch. The bottom shows the indices for a quartic Bézier triangular patch.

21.5.30 Patch Mesh

The `PatchMesh` template defines a mesh of Bézier surface patches. The `vertices` member gives the control points for the patches in the mesh. The `patches` are given as indices into the control `vertices`. The template is open, allowing you to associate arbitary data with the patch mesh.

```
template PatchMesh
{
    <D02C95CC-EDBA-4305-9B5D-1820D7704BBF>
    DWORD nVertices;
    array Vector vertices[nVertices];
    DWORD nPatches;
    array Patch patches[nPatches];
    [...]
}
```

Related Templates

See the `Patch` template on page 745, the `RightHanded` template on page 749, and the `Vector` template on page 751.

21.5.31 Progressive Mesh Attribute Range

The `PMAttributeRange` template gives the range of faces and vertices contained within an attribute subset. Itis used by `ID3DXPMesh` for persistance of progressive meshes. The progressive mesh templates are not defined for use directly by applications and are only given here for completeness.

```
template PMAttributeRange
{
    <917E0427-C61E-4A14-9C64-AFE65F9E9844>
    DWORD iFaceOffset;
    DWORD nFacesMin;
    DWORD nFacesMax;
    DWORD iVertexOffset;
    DWORD nVerticesMin;
    DWORD nVerticesMax;
}
```

Related Templates

See the `PMInfo` template on page 747.

21.5.32 Progressive Mesh Info

The `PMInfo` template gives information about a progressive mesh. It is used by `ID3DXPMesh` for persistance of progressive meshes. The progressive mesh

templates are not defined for use directly by applications and are only given here for completeness. The `PMInfo` template contains `PMAttributeRange` and `PMVSplitRecord` templates as required member data.

```
template PMInfo
{
    <B6C3E656-EC8B-4B92-9B62-681659522947>
    DWORD nAttributes;
    array PMAttributeRange attributeRanges[nAttributes];
    DWORD nMaxValence;
    DWORD nMinLogicalVertices;
    DWORD nMaxLogicalVertices;
    DWORD nVSplits;
    array PMVSplitRecord splitRecords[nVSplits];
    DWORD nAttributeMispredicts;
    array DWORD attributeMispredicts[nAttributeMispredicts];
}
```

Related Templates

See the `PMAttributeRange` template on page 747, and the `PMVSplitRecord` template on page 748.

21.5.33 Progressive Mesh Vertex Split Record

The `PMVSplitRecord` template gives information about the edge collapse transformations used in a progressive mesh. It is used by `ID3DXPMesh` for persistance of progressive meshes. The progressive mesh templates are not defined for use directly by applications and are given here only for completeness.

```
template PMVSplitRecord
{
    <574CCC14-F0B3-4333-822D-93E8A8A08E4C>
    DWORD iFaceCLW;
    DWORD iVlrOffset;
    DWORD iCode;
}
```

Related Templates

See the `PMInfo` template on page 747.

21.5.34 Progressive Mesh

The `ProgressiveMesh` template is a legacy template for use with progressive meshes. It is currently not used, but is listed here for completeness.

```
template ProgressiveMesh
{
    <8A63C360-997D-11D0-941C-0080C80CFA7B>
    [Url <3A23EEA1-94B1-11D0-AB39-0020AF71E433>,
     InlineData <3A23EEA0-94B1-11D0-AB39-0020AF71E433>]
}
```

Related Templates

See the `Url` template on page 751 and the `InlineData` template on page 740.

21.5.35 Property Bag

The `PropertyBag` template defines a collection of properties. Each property has a name and a value, given by a `StringProperty` child node.

```
template PropertyBag
{
    <7F0F21E1-BFE1-11D1-82C0-00A0C9697271>
    [StringProperty <7F0F21E0-BFE1-11D1-82C0-00A0C9697271>]
}
```

Related Templates

See the `StringProperty` template on page 750.

21.5.36 Right Handed

The `RightHanded` template identifies the handedness of the parent `Mesh` node. The `bRightHanded` member has the value one when the mesh is right-handed and the value zero when the mesh is left-handed.

```
template RightHanded
{
    <7F5D5EA0-D53A-11D1-82C0-00A0C9697271>
    DWORD bRightHanded;
}
```

Related Templates

See the `Mesh` template on page 741.

21.5.37 Skin Weights

The `SkinWeights` template associates blending weights with the vertices in a parent `Mesh` node. The `transformNodeName` member gives the name of the coordinate frame of the bone. The `matrixOffset` member gives the transformation

matrix that maps the vertices into the local space of the bone. Each of the `weights` is associated with the vertex whose index is given in the corresponding `vertexIndices` array.

```
template SkinWeights
{
    <6F0D123B-BAD2-4167-A0D0-80224F25FABB>
    STRING transformNodeName;
    DWORD nWeights;
    array DWORD vertexIndices[nWeights];
    array FLOAT weights[nWeights];
    Matrix4x4 matrixOffset;
}
```

Related Templates

See the `Matrix4x4` template on page 741, the `Mesh` template on page 741 and the `XSkinMeshHeader` template on page 752.

21.5.38 String Property

The `StringProperty` template defines an association between a `key` name and its associated string `value`. It is used as a child node of a `PropertyBag` node to create collections of named properties.

```
template StringProperty
{
    <7F0F21E0-BFE1-11D1-82C0-00A0C9697271>
    STRING key;
    STRING value;
}
```

Related Templates

See the `PropertyBag` template on page 749.

21.5.39 Texture Filename

The `TextureFilename` template provides the name of the file containing the texture to be used for textured meshes. It appears as the child node of a `Material` node.

```
template TextureFilename
{
    <A42790E1-7810-11CF-8F52-0040333594A3>
    STRING filename;
}
```

Related Templates

See the `Material` template on page 740.

21.5.40 Timed Float Keys

The `TimedFloatKeys` template associates a set of key frame data with a `time` on the animation timeline. The key frame data is given in the `tfkeys` member. This node appears as a data member of the `AnimationKey` template.

```
template TimedFloatKeys
{
    <F406B180-7B3B-11CF-8F52-0040333594A3>
    DWORD time;
    FloatKeys tfkeys;
}
```

Related Templates

See the `AnimationKey` template on page 732 and the `FloatKeys` template on page 736.

21.5.41 Url

The `Url` template defines a collection of universal resource locator (URL) strings. This template is only used by the deprecated `ProgressiveMesh` template and is listed here only for completeness.

```
template Url
{
    <3A23EEA1-94B1-11D0-AB39-0020AF71E433>
    DWORD nUrls;
    array STRING urls[nUrls];
}
```

Related Templates

See the `ProgressiveMesh` template on page 748.

21.5.42 Vector

The `Vector` template defines a three-dimensional vector. It is also used for three-dimensional points to define vertex positions. It is used by the `Mesh`, `MeshNormals` and `PatchMesh` templates as a data member.

```
template Vector
{
    <3D82AB5E-62DA-11CF-AB39-0020AF71E433>
```

```
    FLOAT x;
    FLOAT y;
    FLOAT z;
}
```

Related Templates

See the `Mesh` template on page 741, the `MeshNormals` template on page 744 and the `PatchMesh` template on page 747.

21.5.43 Vertex Duplication Indices

The `VertexDuplicationIndices` template gives point-representative adjacency information for a mesh. This template will be written to the X file when the D3DX mesh saving routines are passed adjacency information. When reading a mesh from an X file, the data in the template is used to construct the adjacency information returend to the caller. `VertexDuplicationIndices` nodes appear as child nodes of `Mesh` nodes.

The `nIndices` member corresponds to the number of vertices in the parent `Mesh` node. The `nOriginalVertices` member gives the number of original vertices in the mesh. For each vertex in the parent `Mesh` node, the corresponding `indices` array element gives the index of the original vertex.

```
template VertexDuplicationIndices
{
    <B8D65549-D7C9-4995-89CF-53A9A8B031E3>
    DWORD nIndices;
    DWORD nOriginalVertices;
    array DWORD indices[nIndices];
}
```

Related Templates

See the `Mesh` template on page 741.

21.5.44 X Skin Mesh Header

The `XSkinMeshHeader` template gives global information about the skinning applied to a `Mesh` node. The `nBones` member gives the number of bones in the parent `Mesh` node. For each bone in the parent mesh node, there will be one `SkinWeights` node as a child of the mesh.

```
template XSkinMeshHeader
{
    <3CF169CE-FF7C-44AB-93C0-F78F62D172E2>
    WORD nMaxSkinWeightsPerVertex;
    WORD nMaxSkinWeightsPerFace;
```

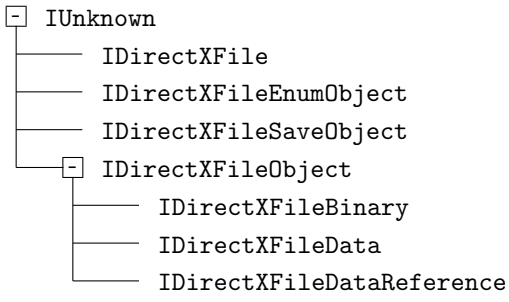

Figure 21.5: .x File Interface Hierarchy

```
    WORD nBones;
}
```

Related Templates

See the `SkinWeights` template on page 749.

21.6 Headers, Libraries and Interfaces

The X file interfaces are not declared in the Direct3D header files. The header file `<dxfile.h>` declares the X file I/O interfaces and the link library `d3dxof.lib` provides exports for `d3dxof.dll`. This DLL is installed with the DirectX runtime and should not be installed by your application. The predefined templates are provided in the header files described in section 21.5.

The interfaces for reading and writing X files are shown according to their inheritence hierarchy in figure 21.5. `IDirectXFile` acts as a factory for creating instances of the `IDirectXFileEnumObject` and `IDirectXFileSaveObject` interfaces. The `IDirectXFileEnumObject` interface is used to read X file data by enumerating the top-level data objects in the file. The `IDirectXFileSaveObject` interface is used to write X file data. The remaining interfaces represent the contents of X files, with the `IDirectXFileObject` interface as a base interface for all the file data. Data nodes are represented by the `IDirectXFileData` interface, binary data is represented by the `IDirectXFileBinary` interface and node references are represented by the `IDirectXFileDataReference` interface.

21.7 IDirectXFile

To read or write scene data with an X file, an instance of `IDirectXFileEnumObject` must be created. The `IDirectXFile` interface has a factory method for creating an instance of the enumeration object. The function `::DirectXFileCreate` is a factory for obtaining an instance of the `IDirectXFile` interface.

As with ::Direct3DCreate9, this function does not require the COM runtime library to be initialized.

```
HRESULT ::DirectXFileCreate(IDirectXFile **result);
```

The IDirectXFile interface is summarized in interface 21.1. The interface is quite simple, with only three methods. The CreateEnumObject method acts as a factory for the IDirectXFileEnumObject interface, while the CreateSaveObject method acts as a factory for the IDirectXFileSaveObject interface. Any necessary data node templates are registered with the X file interfaces with the RegisterTemplates method.

Interface 21.1: Summary of the IDirectXFile interface.

IDirectXFile

Methods	
CreateEnumObject	Create an object that enumerates nodes.
CreateSaveObject	Create an object for saving nodes.
RegisterTemplates	Registers data templates.

```
interface IDirectXFile : IUnknown
{
    // methods
    HRESULT CreateEnumObject(void *source,
                DXFILELOADOPTIONS options,
                IDirectXFileEnumObject **result);
    HRESULT CreateSaveObject(LPCSTR filename,
                DXFILEFORMAT encoding,
                IDirectXFileSaveObject **result);
    HRESULT RegisterTemplates(void *data,
                DWORD size);
};
```

You must register any templates needed by your application before you create the enumeration or save interfaces. The data parameter is a pointer to an X file in memory that contains the template definitions. The file may be in text or binary encoding, but it must be a complete X file with a valid header. If the template definitions are included as text in your application, they should be ANSI text and not Unicode. The size argument gives the size of the block of memory given by data.

The CreateSaveObject method is used to write X file data to the given filename. This method returns an instance of the IDirectXFileSaveObject interface which is used to save the data. The encoding parameter describes the encoding to be used for the new X file. One of the values DXFILEFORMAT_BINARY or DXFILEFORMAT_TEXT should be used. Either of these values can be bitwise

orred with DXFILEFORMAT_COMPRESSED for a compressed file.

```
typedef DWORD DXFILEFORMAT;

#define DXFILEFORMAT_BINARY     0
#define DXFILEFORMAT_TEXT       1
#define DXFILEFORMAT_COMPRESSED 2
```

To read data from an X file after registering any necessary templates, an enumeration object is created with CreateEnumObject to enumerate the top-level data nodes in an X file. The X file can be read from a file, from a Win32 executable resource, from a memory pointer, or from a URL. The options argument describes the location of the X file source and has one of the following values.

```
typedef DWORD DXFILELOADOPTIONS;

#define DXFILELOAD_FROMFILE     0x00L
#define DXFILELOAD_FROMRESOURCE 0x01L
#define DXFILELOAD_FROMMEMORY   0x02L
#define DXFILELOAD_FROMURL      0x08L
```

For data stored in a file, the source parameter gives the name of the file as an ANSI string. For data stored at a URL, the source parameter gives the URL as an ANSI string.

When the data is stored in a Win32 resource, the source parameter points to a DXFILELOADRESOURCE structure. The hModule member provides a handle to the executable module containing the resource, while the lpName and lpType members provide the name and type of the resource within the module. While this structure provides LPCTSTR members, only ANSI resource names and types are supported; the IDirectXFile interface does not provide separate methods for ANSI and Unicode sources. If your application is built for Unicode, you may need to provide your own structure to compensate for the unfortunate use of LPCTSTR here.

```
typedef struct _DXFILELOADRESOURCE
{
    HMODULE hModule;
    LPCTSTR lpName;
    LPCTSTR lpType;
} DXFILELOADRESOURCE, *LPDXFILELOADRESOURCE;
```

When the data is stored in a block of memory, the source parameter points to a DXFILELOADMEMORY structure. The lpMemory member gives a pointer to the X file data in memory and the dSize member gives the size of the memory block.

```
typedef struct _DXFILELOADMEMORY
{
    LPVOID lpMemory;
    DWORD  dSize;
} DXFILELOADMEMORY, *LPDXFILELOADMEMORY;
```

21.8 IDirectXFileEnumObject

The `IDirectXFileEnumObject` interface is summarized in interface 21.2. The `GetNextDataObject` can be used to enumerate all the top-level data objects within the file, returning an instance of `IDirectXFileData` each time it is called. This method enumerates the top-level data objects only; methods from `IDirectXFile` are used to enumerate the nested data objects within a top-level data object. You can obtain an object by name or by instance GUID with the `GetDataObjectByName` or `GetDataObjectById` methods, respectively.

Interface 21.2: Summary of the `IDirectXFileEnumObject` interface.

IDirectXFileEnumObject

Read-Only Properties	
`GetDataObjectById`	The node associated with the id.
`GetDataObjectByName`	The node associated with the name.
`GetNextDataObject`	The next top-level node.

```
interface IDirectXFileEnumObject : IUnknown
{
    // properties
    HRESULT GetNextDataObject(IDirectXFileData **value);
    HRESULT GetDataObjectById(REFGUID guid,
                IDirectXFileData **value);
    HRESULT GetDataObjectByName(LPCSTR name,
                IDirectXFileData **value);
};
```

The `IDirectXFileEnumObject` interface is an example of a forward iterator; it always scans forward through the X file data. There is no way to scan backward, obtain the enumerator's current position or make a copy of the enumerator.

21.9 IDirectXFileObject

The `IDirectXFileObject` interface is the base interface for all data in an X file and is summarized in interface 21.3. This interface is very simple and provides access to the name and identifier GUID properties of a data object.

Interface 21.3: Summary of the `IDirectXFileObject` interface.

IDirectXFileObject

Read-Only Properties	
`GetId`	GUID identifier of the node.
`GetName`	Name of the node.

```
interface IDirectXFileObject : IUnknown
{
    // read-only properties
    HRESULT GetId(LPGUID value);
    HRESULT GetName(LPSTR value, DWORD *size);
};
```

The `GetName` method returns the name associated with the data node, if any. If there is no name associated with the node, then the returned `size` will be zero. `GetName` uses the typical Windows convention of being called once to get the size of the necessary buffer and a second time to copy the data. For example, the following code obtains the necessary size, allocates a buffer of the appropriate size, and then obtains the name property.

```
DWORD size = 0;
THR(xfo->GetName(NULL, &size));
std::vector<char> buffer(size);
THR(xfo->GetName(&buffer[0], &size));
```

The `GetId` method returns the GUID associated with the data node. This is the GUID associated with the instance, not the template GUID. The `value` parameter should point to enough storage for a GUID structure. If no GUID has been associated with the node, then the `value` returned is the `NULL` GUID, whose members are all zero.

21.10 IDirectXFileData

The `IDirectXFileData` interface represents data nodes within the X file. When reading X file data, the properties are used to examine the contents of the data node. When writing X file data, the methods are used to construct a hierarchy of nodes. The interface is summarized in interface 21.4.

Interface 21.4: Summary of the `IDirectXFileData` interface.

IDirectXFileData

Read-Only Properties

`GetData`	The object's associated data
`GetNextObject`	The next child object
`GetType`	The object's template GUID

Methods

`AddBinaryObject`	Adds a binary node as a child
`AddDataObject`	Adds a data node as a child
`AddDataReference`	Adds a data node reference as a child

```
interface IDirectXFileData : IDirectXFileObject
{
    // read-only properties
    HRESULT GetData(LPCSTR member,
                DWORD *size,
                void **result);
    HRESULT GetNextObject(IDirectXFileObject **result);
    HRESULT GetType(const GUID **value);

    // methods
    HRESULT AddBinaryObject(LPCSTR name,
                const GUID *id,
                LPCSTR mime_type,
                void *data,
                DWORD size);
    HRESULT AddDataObject(IDirectXFileData *data);
    HRESULT AddDataReference(LPCSTR name,
                const GUID *id);
};
```

The `GetType` method returns a pointer to the GUID for the template of the node. The template GUID is typically used while reading data to transfer control to a routine for reading the data associated with this particular template.

The `GetNextObject` method returns the `IDirectXFileObject` interface associated with the next child object of the data node. The returned interface may have child objects as well. This method is typically used in a recursive manner to enumerate all child objects within a node hierarchy. The return value is the `IDirectXFileObject` interface for the child node. You can use `QueryInterface` to determine if the child node is a data node, a binary node, or a data reference, as the following example demonstrates.[2]

```
CComPtr<IDirectXFileObject> child;
while (SUCCEEDED(data->GetNextObject(&child)))
{
    CComQIPtr<IDirectXFileData> data(child);
```

[2]This code uses the IID helpers provided in `<rt/iid.h>` in the sample code.

```
    if (data)
    {
        insert_data(self, data);
    }
    else
    {
        CComQIPtr<IDirectXFileDataReference> ref(child);
        if (ref)
        {
            insert_reference(self, ref);
        }
        else
        {
            CComQIPtr<IDirectXFileBinary> binary(child);
            if (binary)
            {
                insert_binary(self, binary);
            }
            else
            {
                THR(E_FAIL);
            }
        }
    }
    child = 0;
}
```

The `GetData` method is used to obtain the `member` data for this node. The `member` parameter gives the name of the member to retrieve, or is `NULL` to obtain all the member data for the node. The `result` parameter returns a pointer to the requested template member data. The data in the returned buffer should never be modified and should be copied into storage provided by the application. The `size` parameter is a pointer to a `DWORD` that will return the size of the returned buffer.

If the X file was in a text encoding, the member data will have been converted into a binary representation suitable for copying directly into a suitable data structure. String data is handled differently from other member data. For string members, the buffer will contain a pointer to the `NULL`-terminated string data instead of the string data itself.

For example, suppose that `xfd` is an `IDirectXFileData` interface representing a `Point3` node according to the template on page 727. The following code reads the data for the point into a `D3DVECTOR` structure. The assertion is added as a defensive programming measure to ensure that the amount of data returned is the amount expected.

```
D3DVECTOR pt;
```

```
DWORD size = 0;
void *buffer = 0;
THR(xfd->GetData(NULL, &size, &buffer));
ATLASSERT(sizeof(D3DVECTOR) == size);
pt = *static_cast<D3DVECTOR *>(buffer);
```

Unfortunately, there is a bug in the implementation of `GetData` when the members contain arrays of variable length items. `GetData` assumes that all the items in an array are the same length and computes the wrong offset into the member data. A workaround is to obtain the data for the entire node instead of by member name. In this case you will need to parse the returned data into the variable-sized array elements yourself.

The `AddDataObject` method is used to add child `data` objects to this data node. The `AddDataReference` method can be used to add a reference to a data node by name or by GUID. One or both of `name` or `id` parameters must be non-`NULL` to identify the referenced node.

The `AddBinaryObject` method is used to add a binary child object to this data node. The `name` and `id` parameters provide optional child node name and GUID for the binary data object. Either of these parameters may be `NULL` if no name or GUID is needed. The `mime_type` parameter gives the Multipurpose Internet Mail Extension (MIME) type of the binary data, as defined by Internet standards. The MIME type is an ASCII string, with "`application/octet-stream`" being the most generic type.[3] The binary data itself is given by the `data` parameter and its size is given by the `size` parameter.

21.11 IDirectXFileBinary

Binary data nodes are represented by the `IDirectXFileBinary` interface, which is summarized in interface 21.5. This method provides two properties for obtaining the MIME type and size of the binary data and a single method for obtaining the data itself.

Interface 21.5: Summary of the `IDirectXFileBinary` interface.

IDirectXFileBinary

Read-Only Properties	
`GetMimeType`	The MIME type of the binary data.
`GetSize`	The size of the binary data.
Methods	
`Read`	Copies binary data from the node.

[3]RFC 2045 and RFC 2046 define MIME types and provide a mechanism for registering new MIME types. A current list of standard MIME types is available at `ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types`.

```
interface IDirectXFileBinary : IDirectXFileObject
{
    // read-only properties
    HRESULT GetMimeType(LPCSTR *value);
    HRESULT GetSize(DWORD *value);

    // methods
    HRESULT Read(void *data,
                 DWORD size,
                 DWORD *size_read);
};
```

The `GetMimeType` method returns the MIME type of the binary data as a pointer to a constant ANSI string. The `GetSize` method returns the size of the binary data.

The `Read` method copies the binary data into the buffer in the `data` parameter. The `size` parameter gives the size of the buffer pointed to by the `data` parameter. If the size of the buffer is larger than the size of the binary data, then the actual amount of data copied is returned in the `size_read` parameter.

21.12 IDirectXFileDataReference

Data node references are represented by the `IDirectXFileDataReference` interface, summarized in interface 21.6. This interface provides a single method, `Resolve`, to resolve a reference and return the corresponding `IDirectXFileData` interface for the referenced node.

Interface 21.6: Summary of the `IDirectXFileDataReference` interface.

IDirectXFileDataReference

Methods	
`Resolve`	Obtain the target of a data node reference.

```
interface IDirectXFileDataReference : IDirectXFileObject
{
    // methods
    HRESULT Resolve(IDirectXFileData **result);
};
```

Unfortunately, there is a bug in the X file parser library that prevents child nodes of a referenced node from being properly accessed.

21.13 IDirectXFileSaveObject

The IDirectXFileSaveObject interface provides a means for creating data nodes, saving templates and saving data nodes to an X file. The interface is summarized in interface 21.7.

Interface 21.7: Summary of the IDirectXFileSaveObject interface.

IDirectXFileSaveObject

Methods	
CreateDataObject	Creates a data node.
SaveData	Write a data node and all its children.
SaveTemplates	Write template definitions.

```
interface IDirectXFileSaveObject : IUnknown
{
    // methods
    HRESULT CreateDataObject(REFGUID template_guid,
                LPCSTR name,
                const GUID *guid,
                DWORD size,
                void *data,
                IDirectXFileData **result);
    HRESULT SaveData(IDirectXFileData *node);
    HRESULT SaveTemplates(DWORD number,
                const GUID **guids);
};
```

The SaveTemplates method can save any templates registered with IDirectX-File when the save object was created. The guids parameter is an array of GUID pointers, giving the template GUIDs to be saved in the file. The number parameter gives the length of the guids array.

The CreateDataObject method is used to create an IDirectXFileData interface. The template for the data object is identified by the template_guid parameter. An optional name and guid for this instance may also be provided; if neither an instance name or GUID is needed, either or both of these parameters may be NULL. The required member data is described by the data and size parameters. The size of the data is given in bytes. String member data is supplied to CreateDataObject as a pointer to the NULL terminated character string. This pointer must remain valid until SaveData is called for the top-level data node containing the string data.

The SaveData method saves the top-level data node, including its member data and all of its nested child nodes, to the file. The AddBinaryObject, Add-DataObject and AddDataReference methods of IDirectXFileData are used to construct the child nodes of the top-level nodes before saving the top-level

nodes to the file.

21.14 Reading Scene Data

Now that we have covered all the relevant interfaces, we can talk more generally about how scene data is read from X files using these interfaces. The easiest way to parse data from the `Mesh` and related templates is to use the D3DX functions for loading meshes from X files.

However, the D3DX functions collapse the node hierarchy into a single flat `ID3DXMesh` object. If you want to reproduce the hierarchy of data in the X file as a hierarchy of data structures in your application, then you need to parse the X file one node at a time using the interfaces described in this chapter.

The SkinnedMesh sample in the SDK provides an example of how to use these interfaces in this manner. The `LoadMeshHierarchy` method in that sample performs the following steps to read an X file:

1. Calls `::DirectXFileCreate` to get an `IDirectXFile` interface.

2. Calls `RegisterTemplates` to register the predefined templates.

3. Calls `CreateEnumObject` to obtain an enumerator over the top-level data nodes in the file.

4. Calls `GetNextDataObject` and `LoadFrames` for each top-level data node.

The `LoadFrames` method in the sample performs the following steps to load the frame hierarchy:

1. Calls `GetType` to obtain the template GUID of the current node.

2. If the node is a `Frame` template, then it builds a data structure for the nested frame and calls `LoadFrames` recursively.

3. If the node is not a `Frame` node, other load methods in the application are called based on the node's template GUID.

This is the general structure for building a hierarchical data structure in your application that mimics the hierarchical structure of an X file. The organization of the code will be similar to a recursive descent parser for a simple language; the difference is that the X file interfaces are performing the low-level parsing of the data and your application need only worry about mapping the data contained within the X file nodes to the data structure used by your application.

The SDK sample framework contains the source code for the classes `CD3DMesh`, `CD3DFrame` and `CD3DFile`. These classes support the loading of a hierarchy of frames from an X file. They are discussed in more detail in appendix B.

21.15 Writing Scene Data

Reading data from an X file is a little simpler than writing data. The SDK does not include a sample of writing X file data. However, the sample program for this chapter shows the details of writing an X file.

The general approach is the reverse of what is done to build a recursive data structure from an X file's nodes. The existing recursive data structure is traversed, building an `IDirectXFileData` object for each node in the data structure with `CreateDataObject`. The `AddDataObject` method is used to add child nodes to their parent node, building a hierarchy of `IDirectXFileData` interfaces that mimics the hierarchy in your application's data structure. Once the hierarchy has been built, `SaveData` can be called to save the top-level model and all its child objects to the X file.

An `IDirectXFileData` object is created from a buffer of data. The buffer must be prepared so that its memory layout follows the memory layout of the member data in the corresponding template for the node. In the chapter sample, the standard vector class is used to create a buffer of `DWORD`s to hold node data. A vector of strings is also used to keep string data pointers valid until `SaveData` has been called.

Strings written to X files in text mode will need to have any backslash (\) characters doubled before `SaveData` is called. This is due to the way that X files are written in text mode. This is particularly important for the `TextureFilename` nodes when the filename contains a path. X files in the binary encoding do not need this adjustment to string member data.

21.16 Exporting and Conversion

The SDK includes source code and precompiled binaries for export plugins for several popular modeling packages. Exporters are provided for 3D Studio Max (version 3.0 and 4.0) and Maya (versions 2.5, 3.0 and 4.0). These exporters are unsupported by Microsoft. They are provided in source code form so that you may customize them to suit your specific needs. You can also use them as a starting point for an exporter of your own. Eric DeBrosse, Microsoft Visual Basic MVP, has written several articles about using and improving the supplied exporters in the SDK. They can be viewed on his web site `mvps.org/vbdx/`.

An exporter for the Milkshape free modeler is available at `www.cfxweb.net/files/Detailed/1093.shtml`. PandaSoft provides an alternative 3DS Max exporter at `www.pandasoft.demon.co.uk/directxmax4.htm`. Other exporters can be found by searching the web for "DirectX exporter".

An alternative to using an exporter plugin for a modeler is to use a 3D model file converter. There are a variety of commercial and freely available file conversion utilities. The SDK includes an unsupported file converter called `conv3ds` for converting 3D Studio `3ds` files to the X file format. See section A.9 for the details of using this tool. A freely available conversion tool called Crossroads 3D and is available on the web at `home.europa.com/~keithr/crossroads/`.

21.17 Example X Files

The best way to learn about the structure of X files is to edit them in a text editor and view them in the MeshView utility included with the SDK. Unfortunately, MeshView isn't very helpful when you have an error in your X file, but the parsing interfaces generally output a diagnostic about any parse errors to the debug output stream when you run your program under the debugger.

The simple examples in this section use the predefined templates provided with the SDK. The examples are given in the text encoding, with comments included to explain the data members. This first example creates a triangle with only positions at the vertices using the `Mesh` template.

```
xof 0303txt 0032

Mesh
{
    3;              // number of vertices
    0.0; 0.0; 0.0;, // v0
    0.5; 1.0; 0.0;, // v1
    1.0; 0.0; 0.0;; // v2
    1;              // number of faces
    3;              // 3 vertices in the face
    0, 1, 2;;       // indices for face 0
}
```

We can enhance the shading of the triangle if we add normals for the vertices of the triangle with the `MeshNormals` template.

```
xof 0303txt 0032

Mesh
{
    3;
    0.0; 0.0; 0.0;,
    0.5; 1.0; 0.0;,
    1.0; 0.0; 0.0;;
    1;
    3;
    0, 1, 2;;

    MeshNormals
    {
        3;
        -0.7071; 0;      -0.7071;,  // n0
         0.0;    0.7071; -0.7071;,  // n1
         0.7071; 0;      -0.7071;;  // n2
        1;
```

```
        3;
        0, 1, 2;;
    }
}
```

We can add colors at each of the vertices with the `MeshVertexColors` template to obtain a more interesting appearance. Note that each color is terminated with two semi-colons. The first terminates the alpha component of the `ColorRGBA` member and the second terminates the `IndexedColor` member.

```
xof 0303txt 0032

Mesh
{
    3;
    0.0; 0.0; 0.0;,
    0.5; 1.0; 0.0;,
    1.0; 0.0; 0.0;;
    1;
    3;
    0, 1, 2;;

    MeshNormals
    {
        3;
        -0.7071; 0;      -0.7071;,
         0.0;    0.7071; -0.7071;,
         0.7071; 0;      -0.7071;;
        1;
        3;
        0, 1, 2;;
    }

    MeshVertexColors
    {
        3;
        // vertex index, RGBA color
        0; 1.0; 0.0; 0.0; 1.0;;,
        1; 0.0; 1.0; 0.0; 1.0;;,
        2; 0.0; 0.0; 1.0; 1.0;;;
    }
}
```

The SDK includes many additional examples of X files in the Media subdirectory.

21.18 rt_MakeScene Sample Program

The rt_MakeScene sample creates a simple scene hierarchy in memory and demonstrates how to write a scene hierarchy to an X file in either the text or binary encoding. The X file may also be compressed. Shown here are the "interesting" parts of the sample; the full source code is included in the accompanying sample code.

This sample program uses the CD3DMesh, CD3DFile and CD3DFrame classes from the SDK sample framework. These classes are extended in frame.h and frame.cpp to allow a simple scene to be constructed dynamically and to support saving a hierarchy to an X file.

The first listing, rt_MakeScene.cpp, shows how the classes were extended to support the creation of a simple scene hierarchy directly in code. The sample supports three different kinds of simple scenes: a "temple" scene constructed from simple shape meshes created with D3DX and two regular decompositions of space that approximate fractal solids. The "Menger" decomposition produces the fractal known as the "Menger sponge" when subdivided by an infinite number of levels. The "Resch" scene is similar, but uses a different spatial decomposition due to Ron Resch. The functions init_sponge and init_resch, which construct the spatial decomposition scenes, are not shown for brevity, but their full source is included in the sample.

The code for the temple scene shows the use of the new add method on extended frame class. The member variable m_root is declared to be an instance of the frame class. The arguments to add are the name of the node, the ID3DXMesh interface to be added, the transformation matrix applied to the mesh to position it relative to the parent node and the material for the mesh.

The next listing, frame.h, gives the declarations for the vertex structures, the c_material helper class that extends D3DMATERIAL9, and the c_frame class that extends CD3DFrame. The public interface of c_frame provides the add methods for building the scene and the save method for writing a frame and its children to an X file. The private portion of c_frame contains the DWORD and string buffers used for building IDirectXFileData nodes. The private methods are used to handle the details of saving a mesh.

The final listing, frame.cpp, shows the details of constructing the nodes in memory and building a node hierarchy that reflects the hierarchy of CD3DFrame objects. Lines 1–162 contain the code for building scenes and the remaining code contains the details of saving scenes to an X file. The create_data_object helper function simplifies the construction of an IDirectXFileData object from a buffer as a vector of DWORDs. The overloaded push functions simplify the construction of the memory image of a node.

The save_plain_mesh and save_textured_mesh functions are called by save_mesh depending on the FVF code of the underlying ID3DXMesh object. They create the necessary IDirectXFileData objects to represent plain and textured meshes with the Mesh, MeshNormals, MeshMaterialList, Material and MeshTextureCoords templates. Each function works by building the necessary images, calling CreateDataObject through the helper function, and assembling

the data objects into the appropriate hierarchy with `AddDataObject`.

Finally, the `save` method is present in two overloaded forms. The first form handles the recursive traversal of the scene, constructing `Frame` and `FrameTransformMatrix` templates as necessary. The second overloaded form handles the details of creating the save object with the appropriate encoding flags and calling the first overloaded form to perform the recursive traversal.

Listing 21.1: rt_MakeScene.cpp: X file output.

```
1   ////////////////////////////////////////////////////////////
2   // File: rt_MakeScene.cpp
3   //
4   // Create a simple scene hierarchy and demonstrate saving that
5   // hierarchy to an X file.
6   //
7   #include <sstream>
8
9   #define STRICT
10  #include <windows.h>
11
12  #include <D3DX9.h>
13
14  #include "dxutil.h"
15  #include "d3denumeration.h"
16  #include "d3dsettings.h"
17  #include "D3DApp.h"
18  #include "D3DFont.h"
19  #include "rt_MakeScene.h"
20
21  #include "rt/hr.h"
22  #include "rt/mat.h"
23  #include "rt/mesh.h"
24  #include "rt/misc.h"
25  #include "rt/tstring.h"
26
27  ////////////////////////////////////////////////////////////
28  // CMyD3DApplication::init_scene
29  //
30  // Create the scene based on the current selection.  Construct
31  // the "temple" scene directly using D3DX primitives.  For
32  // the Menger and Resch scenes, call auxiliary methods that
33  // generate the triangles directly.
34  //
35  void
36  CMyD3DApplication::init_scene()
37  {
38      const c_material yellow(1, 1, 0);
39      const c_material blue(0, 0.5f, 1);
40      const c_material green(0, 1, 0);
41      const D3DXMATRIX rot_x_pi2 = rt::mat_rot_x(D3DX_PI*0.5f);
42
43      switch (m_scene)
44      {
```

```
45      case ES_TEMPLE:
46          {
47              CComPtr<ID3DXMesh> mesh;
48              rt::dx_buffer<DWORD> adj;
49              THR(::D3DXCreateBox(m_pd3dDevice, 10, 0.1f, 10,
50                  &mesh, &adj));
51              THR(mesh->OptimizeInplace(D3DXMESHOPT_VERTEXCACHE,
52                  adj, NULL, NULL, NULL));
53              m_root.add(_T("floor"), mesh,
54                  rt::mat_trans(0, -0.05f, 0), yellow);
55              m_root.add(_T("ceiling"), mesh,
56                  rt::mat_trans(0, 10.05f, 0), yellow);
57          }
58          {
59              CComPtr<ID3DXMesh> mesh;
60              rt::dx_buffer<DWORD> adj;
61              THR(::D3DXCreateBox(m_pd3dDevice, 10, 10.2f, 0.1f,
62                  &mesh, &adj));
63              THR(mesh->OptimizeInplace(D3DXMESHOPT_VERTEXCACHE,
64                  adj, NULL, NULL, NULL));
65              m_root.add(_T("back"), mesh, rt::mat_trans(0, 5, 5),
66                  yellow);
67          }
68          {
69              CComPtr<ID3DXMesh> mesh;
70              rt::dx_buffer<DWORD> adj;
71              THR(::D3DXCreateCylinder(m_pd3dDevice, 0.5f, 0.5f,
72                  10, 20, 40, &mesh, &adj));
73              THR(mesh->OptimizeInplace(D3DXMESHOPT_VERTEXCACHE,
74                  adj, NULL, NULL, NULL));
75              for (int i = 0; i < 6; i++)
76              {
77                  rt::tostringstream name;
78                  name << _T("column_E") << (i+1) << std::ends;
79                  m_root.add(name.str().c_str(), mesh,
80                      rt::mat_trans(4, -4.f + i*8.f/5.f, -5)*
81                      rot_x_pi2, blue);
82                  name.seekp(0, std::ios::beg);
83                  name << _T("column_W") << (i+1) << std::ends;
84                  m_root.add(name.str().c_str(), mesh,
85                      rt::mat_trans(-4, -4.f + i*8.f/5.f, -5)*
86                      rot_x_pi2, blue);
87              }
88          }
89          {
90              CComPtr<ID3DXMesh> mesh;
```

```
91              rt::dx_buffer<DWORD> adj;
92              THR(::D3DXCreateTorus(m_pd3dDevice, 0.1f, 1.f,
93                  20, 40, &mesh, &adj));
94              THR(mesh->OptimizeInplace(D3DXMESHOPT_VERTEXCACHE,
95                  adj, NULL, NULL, NULL));
96              m_root.add(_T("object"), mesh, rot_x_pi2*
97                  rt::mat_trans(0, 0.5, 0), green);
98          }
99          break;
100
101     case ES_MENGER_SPONGE_0:
102     case ES_MENGER_SPONGE_1:
103     case ES_MENGER_SPONGE_2:
104     case ES_MENGER_SPONGE_3:
105         init_sponge(UINT(m_scene - ES_MENGER_SPONGE_0));
106         break;
107
108     case ES_RESCH_0:
109     case ES_RESCH_1:
110     case ES_RESCH_2:
111     case ES_RESCH_3:
112     case ES_RESCH_4:
113     case ES_RESCH_5:
114         init_resch(UINT(m_scene - ES_RESCH_0));
115         break;
116
117     default:
118         ATLASSERT(false);
119     }
120 }
```

Listing 21.2: `frame.h`: X file output.

```
#if !defined(RT_FRAME_H)
////////////////////////////////////////////////////////////////
#define RT_FRAME_H

#include <string>
#include <vector>

#include <d3dx9mesh.h>

#include "d3dfile.h"
#include "d3dutil.h"

#include "rt/tstring.h"

////////////////////////////////////////////////////////////////
// s_plain_vertex
//
// A plain vertex with a position and normal.
//
struct s_plain_vertex
{
    s_plain_vertex(float x = 0, float y = 0, float z = 0,
                   float nx = 0, float ny = 0, float nz = 0)
        : m_pos(x, y, z),
        m_normal(nx, ny, nz)
    {}

    D3DXVECTOR3 m_pos;
    D3DXVECTOR3 m_normal;

    static const DWORD FVF;
};

////////////////////////////////////////////////////////////////
// s_textured_vertex
//
// A vertex with a position, normal and 2D texture coordinates.
//
struct s_textured_vertex : public s_plain_vertex
{
    s_textured_vertex(float x = 0, float y = 0, float z = 0,
                      float nx = 0, float ny = 0, float nz = 0,
                      float u = 0, float v = 0)
        : s_plain_vertex(x, y, z, nx, ny, nz),
```

```
45          m_tex(u, v)
46      {}
47
48      D3DXVECTOR2 m_tex;
49
50      static const DWORD FVF;
51  };
52
53  //////////////////////////////////////////////////////////////////
54  // c_material
55  //
56  // Helper class for initializing D3DMATERIAL9 and providing a
57  // texture filename.
58  //
59  class c_material : public D3DMATERIAL9
60  {
61  public:
62      c_material()
63          : D3DMATERIAL9(),
64          m_texture_filename()
65      {
66          ::ZeroMemory(static_cast<D3DMATERIAL9 *>(this),
67              sizeof(D3DMATERIAL9));
68      }
69
70      c_material(const c_material &rhs)
71          : D3DMATERIAL9(rhs),
72          m_texture_filename(rhs.m_texture_filename)
73      {}
74
75      c_material(float red, float green, float blue, float alpha = 1)
76          : D3DMATERIAL9(),
77          m_texture_filename()
78      {
79          ::D3DUtil_InitMaterial(*this, red, green, blue, alpha);
80      }
81
82      c_material(const rt::tstring &filename, float red = 1,
83                 float green = 1, float blue = 1, float alpha = 1)
84          : D3DMATERIAL9(),
85          m_texture_filename(filename)
86      {
87          ::D3DUtil_InitMaterial(*this, red, green, blue, alpha);
88      }
89
90      const rt::tstring &texture() const
```

```
91      {
92          return m_texture_filename;
93      }
94
95  private:
96      rt::tstring m_texture_filename;
97  };
98
99  ////////////////////////////////////////////////////////////
100 // c_frame
101 //
102 // A helper class to assist in building and saving scenes.
103 // It extends the CD3DFrame class in the SDK sample framework.
104 //
105 class c_frame : public CD3DFrame
106 {
107 public:
108     c_frame(const rt::tstring &name)
109         : CD3DFrame(name.c_str())
110     {
111     }
112     virtual ~c_frame()
113     {
114     }
115
116     void add(const rt::tstring &name,
117              ID3DXMesh *mesh,
118              const D3DXMATRIX &matrix,
119              const c_material &material);
120
121     void add(const rt::tstring &name,
122              ID3DXMesh *mesh,
123              const D3DXMATRIX &matrix,
124              std::vector<c_material> &materials);
125
126     void save(const rt::tstring &filename,
127               bool binary,
128               bool compressed);
129
130 private:
131     LPCSTR save_string(const rt::tstring &value) const
132     {
133         USES_CONVERSION;
134         s_strings.push_back(T2CA(value.c_str()));
135         return s_strings.back().c_str();
136     }
```

```
137
138       void add_aux(const rt::tstring &name, ID3DXMesh *mesh,
139                    const D3DXMATRIX &matrix, CD3DMesh *child_mesh);
140
141       void save(ID3DXFileSaveObject *so,
142                 ID3DXFileSaveData *file_node,
143                 CD3DFrame *node);
144
145       void save_mesh(ID3DXFileSaveObject *so,
146                      ID3DXFileSaveData *parent,
147                      CD3DMesh *mesh);
148
149       void save_plain_mesh(ID3DXFileSaveObject *so,
150                      ID3DXFileSaveData *parent,
151                      CD3DMesh *mesh);
152
153       void save_textured_mesh(ID3DXFileSaveObject *so,
154                      ID3DXFileSaveData *parent,
155                      CD3DMesh *mesh);
156
157       static std::vector<DWORD> s_buffer;
158       static std::vector<std::string> s_strings;
159   };
160
161   #endif
```

Listing 21.3: `frame.cpp`: X file output.

```
1   #include <vector>
2   #include <sstream>
3
4   #include <atlbase.h>
5
6   #include <d3dx9.h>
7   #include <rmxfguid.h>
8
9   #include "frame.h"
10  #include "templates.h"
11
12  #include "rt/hr.h"
13  #include "rt/media.h"
14  #include "rt/mesh.h"
15  #include "rt/misc.h"
16  #include "rt/states.h"
17  #include "rt/tstring.h"
18
19  //////////////////////////////////////////////////////////////
20  // Static member initializers
21  //
22  std::vector<DWORD> c_frame::s_buffer;
23  std::vector<std::string> c_frame::s_strings;
24
25  //////////////////////////////////////////////////////////////
26  // FVF static member initializers
27  //
28  const DWORD s_plain_vertex::FVF = D3DFVF_XYZ | D3DFVF_NORMAL;
29
30  const DWORD s_textured_vertex::FVF = s_plain_vertex::FVF |
31      D3DFVF_TEX1 | D3DFVF_TEXCOORDSIZE2(0);
32
33  //////////////////////////////////////////////////////////////
34  // s_material
35  //
36  // This structure represents the layout of a Material template,
37  // which differs from D3DMATERIAL9.  The constructor initializes
38  // the structure from a D3DMATERIAL9.
39  //
40  struct s_material
41  {
42      s_material(const D3DMATERIAL9 &m)
43          : m_diffuse(m.Diffuse.r, m.Diffuse.g,
44                      m.Diffuse.b, m.Diffuse.a),
```

```
45          m_power(m.Power),
46          m_specular(m.Specular.r, m.Specular.g, m.Specular.b),
47          m_emissive(m.Emissive.r, m.Emissive.g, m.Emissive.b)
48      {
49      }
50      D3DXVECTOR4 m_diffuse;
51      float m_power;
52      D3DXVECTOR3 m_specular;
53      D3DXVECTOR3 m_emissive;
54  };
55
56  //////////////////////////////////////////////////////////////
57  // c_frame::add
58  //
59  // Add a named mesh to the frame, positioned relative to the
60  // frame by the given matrix and drawn with the given materials.
61  //
62  void
63  c_frame::add(const rt::tstring &name,
64               ID3DXMesh *mesh,
65               const D3DXMATRIX &matrix,
66               std::vector<c_material> &materials)
67  {
68      CD3DMesh *child_mesh = new CD3DMesh(name.c_str());
69      mesh->AddRef();
70      child_mesh->m_pSysMemMesh = mesh;
71      child_mesh->m_bUseMaterials = TRUE;
72      child_mesh->m_dwNumMaterials = materials.size();
73      child_mesh->m_pTextures =
74          new LPDIRECT3DTEXTURE9[materials.size()];
75      child_mesh->m_pMaterials =
76          new D3DMATERIAL9[materials.size()];
77
78      // Copy each material and create its texture
79      CComPtr<IDirect3DDevice9> device;
80      THR(mesh->GetDevice(&device));
81      for (UINT i = 0; i < materials.size(); i++)
82      {
83          child_mesh->m_pMaterials[i] = materials[i];
84          child_mesh->m_pMaterials[i].Ambient =
85              child_mesh->m_pMaterials[i].Diffuse;
86          child_mesh->m_pTextures[i] = NULL;
87          if (materials[i].texture().size())
88          {
89              THR(::D3DXCreateTextureFromFile(device,
90                  rt::find_media(materials[i].texture()).c_str(),
```

```
91                  &child_mesh->m_pTextures[i]));
92          }
93      }
94
95      add_aux(name, mesh, matrix, child_mesh);
96  }
97
98  //////////////////////////////////////////////////////////////
99  // c_frame::add
100 //
101 // Add a named mesh to the frame, positioned relative to the
102 // frame by the given matrix and drawn with the given amterial.
103 //
104 void
105 c_frame::add(const rt::tstring &name,
106              ID3DXMesh *mesh,
107              const D3DXMATRIX &matrix,
108              const c_material &material)
109 {
110     CD3DMesh *child_mesh = new CD3DMesh(name.c_str());
111     mesh->AddRef();
112     child_mesh->m_pSysMemMesh = mesh;
113     child_mesh->m_bUseMaterials = TRUE;
114     child_mesh->m_dwNumMaterials = 1;
115     child_mesh->m_pTextures = new LPDIRECT3DTEXTURE9[1];
116     child_mesh->m_pMaterials = new D3DMATERIAL9[1];
117     child_mesh->m_pTextures[0] = NULL;
118     child_mesh->m_pMaterials[0] = material;
119
120     add_aux(name, mesh, matrix, child_mesh);
121 }
122
123 //////////////////////////////////////////////////////////////
124 // c_frame::add_aux
125 //
126 // Helper method used by c_frame::add
127 //
128 void
129 c_frame::add_aux(const rt::tstring &name, ID3DXMesh *mesh,
130                  const D3DXMATRIX &matrix, CD3DMesh *child_mesh)
131 {
132     c_frame *child = new c_frame(name);
133     child->m_mat = matrix;
134
135     const UINT len = NUM_OF(child->m_strName)-1;
136     _tcsncpy(child->m_strName, name.c_str(), len);
```

```
137       child->m_strName[len] = 0;
138       child->m_pMesh = child_mesh;
139       if (m_pChild)
140       {
141           CD3DFrame *sibling = m_pChild;
142
143           while (sibling->m_pNext)
144           {
145                sibling = sibling->m_pNext;
146           }
147           sibling->m_pNext = child;
148       }
149       else
150       {
151           m_pChild = child;
152       }
153   }
154
155   ////////////////////////////////////////////////////////////////
156   // create_data_object
157   //
158   // Helper function to handle the details of creating a data
159   // object from a std::vector<DWORD> buffer.
160   //
161   void
162   create_data_object(ID3DXFileSaveObject *so, REFGUID guid,
163                      CD3DMesh *mesh, LPCTSTR suffix,
164                      const std::vector<DWORD> &buffer,
165                      ID3DXFileSaveData **result)
166   {
167           USES_CONVERSION;
168       THR(so->AddDataObject(guid,
169           T2CA((rt::tstring(mesh->m_strName) + suffix).c_str()),
170                   NULL, buffer.size()*sizeof(DWORD),
171           const_cast<DWORD *>(&buffer[0]), result));
172   }
173
174   void
175   create_data_object(ID3DXFileSaveData *parent, REFGUID guid,
176                                      CD3DMesh *mesh, LPCTSTR suffix,
177                                      const std::vector<DWORD> &buffer,
178                                      ID3DXFileSaveData **result)
179   {
180           USES_CONVERSION;
181           THR(parent->AddDataObject(guid,
182                   T2CA((rt::tstring(mesh->m_strName) + suffix).c_str()),
```

```
183                 NULL, buffer.size()*sizeof(DWORD), &buffer[0], result));
184 }
185
186 ////////////////////////////////////////////////////////////
187 // push_vector, push_indices
188 //
189 // Helper functions to push data onto the DWORD buffer
190 //
191 void
192 push_vector(std::vector<DWORD> &buffer, const D3DXVECTOR2 &v)
193 {
194     buffer.push_back(rt::float_dword(v.x));
195     buffer.push_back(rt::float_dword(v.y));
196 }
197
198 void
199 push_vector(std::vector<DWORD> &buffer, const D3DXVECTOR3 &v)
200 {
201     buffer.push_back(rt::float_dword(v.x));
202     buffer.push_back(rt::float_dword(v.y));
203     buffer.push_back(rt::float_dword(v.z));
204 }
205
206 void
207 push_indices(std::vector<DWORD> &buffer, const WORD *indices)
208 {
209     buffer.push_back(3);
210     buffer.push_back(indices[0]);
211     buffer.push_back(indices[1]);
212     buffer.push_back(indices[2]);
213 }
214
215 ////////////////////////////////////////////////////////////
216 // c_frame::save_plain_mesh
217 //
218 // Saves a mesh with plain vertices.
219 //
220 void
221 c_frame::save_plain_mesh(ID3DXFileSaveObject *so,
222                          ID3DXFileSaveData *parent,
223                          CD3DMesh *mesh)
224 {
225     CComPtr<ID3DXMesh> dxmesh = mesh->m_pSysMemMesh;
226     ATLASSERT(s_plain_vertex::FVF == dxmesh->GetFVF());
227     const UINT num_vertices = dxmesh->GetNumVertices();
228     const UINT num_faces = dxmesh->GetNumFaces();
```

```
229
230         CComPtr<ID3DXFileSaveData> mesh_node;
231         {
232             std::vector<DWORD> normals;
233             s_buffer.clear();
234             const UINT size = 1 + num_vertices*3 + 1 + num_faces*4;
235             s_buffer.reserve(size);
236             normals.reserve(size);
237
238             s_buffer.push_back(num_vertices);
239             normals.push_back(num_vertices);
240             {
241                 const rt::mesh_vertex_lock<s_plain_vertex>
242                     lock(dxmesh, D3DLOCK_READONLY);
243                 const s_plain_vertex *verts = lock.data();
244
245                 for (UINT i = 0; i < num_vertices; i++)
246                 {
247                     push_vector(s_buffer, verts[i].m_pos);
248                     push_vector(normals, verts[i].m_normal);
249                 }
250             }
251
252             s_buffer.push_back(num_faces);
253             normals.push_back(num_faces);
254             {
255                 const rt::mesh_index_lock<WORD>
256                     lock(dxmesh, D3DLOCK_READONLY);
257                 const WORD *indices = lock.data();
258
259                 for (UINT i = 0; i < num_faces; i++)
260                 {
261                     push_indices(s_buffer, &indices[i*3]);
262                     push_indices(normals, &indices[i*3]);
263                 }
264             }
265
266                 if (parent)
267                 {
268                         create_data_object(parent, TID_D3DRMMesh, mesh,
269                                 _T("_mesh"), s_buffer, &mesh_node);
270                 }
271                 else
272                 {
273                         create_data_object(so, TID_D3DRMMesh, mesh,
274                     _T("_mesh"), s_buffer, &mesh_node);
```

```
275                 }
276
277             CComPtr<ID3DXFileSaveData> normals_node;
278             create_data_object(mesh_node, TID_D3DRMMeshNormals,
279                             mesh, _T("_normals"), normals, &normals_node);
280         }
281         {
282             s_buffer.clear();
283             s_buffer.push_back(mesh->m_dwNumMaterials);
284             s_buffer.push_back(num_faces);
285             UINT i;
286             {
287                 const rt::mesh_attribute_lock
288                     lock(dxmesh, D3DLOCK_READONLY);
289                 const DWORD *attributes = lock.data();
290
291                 for (i = 0; i < num_faces; i++)
292                 {
293                     s_buffer.push_back(attributes[i]);
294                 }
295             }
296
297             CComPtr<ID3DXFileSaveData> matlist_node;
298             create_data_object(mesh_node, TID_D3DRMMeshMaterialList,
299                             mesh, _T("_materials"), s_buffer, &matlist_node);
300
301                 USES_CONVERSION;
302             for (i = 0; i < mesh->m_dwNumMaterials; i++)
303             {
304                 CComPtr<ID3DXFileSaveData> material_node;
305                 rt::tostringstream node_name;
306                 node_name << mesh->m_strName << _T("_material")
307                     << i << std::ends;
308                 THR(matlist_node->AddDataObject(TID_D3DRMMaterial,
309                     T2CA(node_name.str().c_str()), NULL,
310                     sizeof(s_material),
311                     &s_material(mesh->m_pMaterials[i]),
312                     &material_node));
313             }
314         }
315     }
316
317     ////////////////////////////////////////////////////////////
318     // c_frame::save_textured_mesh
319     //
320     // Saves a mesh with textured vertices.
```

```
321 //
322 void
323 c_frame::save_textured_mesh(ID3DXFileSaveObject *so,
324                             ID3DXFileSaveData *parent,
325                             CD3DMesh *mesh)
326 {
327         USES_CONVERSION;
328     CComPtr<ID3DXMesh> dxmesh = mesh->m_pSysMemMesh;
329     ATLASSERT(s_textured_vertex::FVF == dxmesh->GetFVF());
330     const UINT num_vertices = dxmesh->GetNumVertices();
331     const UINT num_faces = dxmesh->GetNumFaces();
332
333     CComPtr<ID3DXFileSaveData> mesh_node;
334     {
335         std::vector<DWORD> normals;
336         std::vector<DWORD> uvs;
337         s_buffer.clear();
338         const UINT size = 1 + num_vertices*3 + 1 + num_faces*4;
339         s_buffer.reserve(size);
340         normals.reserve(size);
341         uvs.reserve(size);
342
343         s_buffer.push_back(num_vertices);
344         normals.push_back(num_vertices);
345         uvs.push_back(num_vertices);
346         {
347             const rt::mesh_vertex_lock<s_textured_vertex>
348                 lock(dxmesh, D3DLOCK_READONLY);
349             const s_textured_vertex *verts = lock.data();
350
351             for (UINT i = 0; i < num_vertices; i++)
352             {
353                 push_vector(s_buffer, verts[i].m_pos);
354                 push_vector(normals, verts[i].m_normal);
355                 push_vector(uvs, verts[i].m_tex);
356             }
357         }
358
359         s_buffer.push_back(num_faces);
360         normals.push_back(num_faces);
361         uvs.push_back(num_faces);
362         {
363             const rt::mesh_index_lock<WORD>
364                 lock(dxmesh, D3DLOCK_READONLY);
365             const WORD *indices = lock.data();
366
```

```
367             for (UINT i = 0; i < num_faces; i++)
368             {
369                 push_indices(s_buffer, &indices[i*3]);
370                 push_indices(normals, &indices[i*3]);
371                 push_indices(uvs, &indices[i*3]);
372             }
373         }
374 
375                 if (parent)
376                 {
377                         create_data_object(parent, TID_D3DRMMesh, mesh,
378                                 _T("_mesh"), s_buffer, &mesh_node);
379                 }
380                 else
381                 {
382                         create_data_object(so, TID_D3DRMMesh, mesh,
383                                 _T("_mesh"), s_buffer, &mesh_node);
384                 }
385 
386         CComPtr<ID3DXFileSaveData> normals_node;
387         create_data_object(mesh_node, TID_D3DRMMeshNormals, mesh,
388             _T("_normals"), normals, &normals_node);
389 
390         CComPtr<ID3DXFileSaveData> uvs_node;
391         create_data_object(mesh_node, TID_D3DRMMeshTextureCoords,
392             mesh, _T("_texcoords"), uvs, &uvs_node);
393     }
394     {
395         s_buffer.clear();
396         s_buffer.push_back(mesh->m_dwNumMaterials);
397         s_buffer.push_back(num_faces);
398 
399         UINT i;
400         {
401             const rt::mesh_attribute_lock
402                 lock(dxmesh, D3DLOCK_READONLY);
403             const DWORD *attributes = lock.data();
404 
405             for (i = 0; i < num_faces; i++)
406             {
407                 s_buffer.push_back(attributes[i]);
408             }
409         }
410 
411         CComPtr<ID3DXFileSaveData> matlist_node;
412         create_data_object(mesh_node, TID_D3DRMMeshMaterialList,
```

```
413             mesh, _T("_materials"), s_buffer, &matlist_node);
414
415         for (i = 0; i < mesh->m_dwNumMaterials; i++)
416         {
417             CComPtr<ID3DXFileSaveData> material_node;
418             rt::tostringstream node_name;
419             node_name << mesh->m_strName << _T("_material")
420                 << i << std::ends;
421             THR(matlist_node->AddDataObject(TID_D3DRMMaterial,
422                                 T2CA(node_name.str().c_str()), NULL,
423                 sizeof(s_material),
424                 &s_material(mesh->m_pMaterials[i]),
425                 &material_node));
426
427             CComPtr<ID3DXFileSaveData> texturename_node;
428             node_name.seekp(0);
429             node_name << mesh->m_strName << _T("_texture_name")
430                 << i << std::ends;
431
432             // WARNING: this string needs to live until
433             // ID3DXFileSaveObject::SaveData is called
434             LPCSTR data =
435                 save_string(mesh->m_pTextureFilenames[i]);
436             THR(material_node->AddDataObject(
437                                 TID_D3DRMTextureFilename,
438                 T2CA(node_name.str().c_str()), NULL, sizeof(void *),
439                 &data, &texturename_node));
440         }
441     }
442 }
443
444 ////////////////////////////////////////////////////////////
445 // c_frame::save_mesh
446 //
447 // Examine the FVF of the given mesh and call the appropriate
448 // save mesh method.
449 //
450 void
451 c_frame::save_mesh(ID3DXFileSaveObject *so,
452                    ID3DXFileSaveData *parent,
453                    CD3DMesh *mesh)
454 {
455     const DWORD fvf = mesh->m_pSysMemMesh->GetFVF();
456     if (s_plain_vertex::FVF == fvf)
457     {
458         save_plain_mesh(so, parent, mesh);
```

```
459         }
460         else if (s_textured_vertex::FVF == fvf)
461         {
462             save_textured_mesh(so, parent, mesh);
463         }
464         else
465         {
466             ATLASSERT(false);
467         }
468     }
469
470     ////////////////////////////////////////////////////////////
471     // c_frame::save
472     //
473     // Recursively traverse this frame to save all meshes and frames.
474     //
475     void
476     c_frame::save(ID3DXFileSaveObject *so, ID3DXFileSaveData *parent,
477                   CD3DFrame *frame)
478     {
479             USES_CONVERSION;
480         CComPtr<ID3DXFileSaveData> frame_node;
481             if (parent)
482             {
483                     THR(parent->AddDataObject(TID_D3DRMFrame,
484                             T2CA(frame->m_strName), NULL, 0, NULL, &frame_node));
485             }
486             else
487             {
488                     THR(so->AddDataObject(TID_D3DRMFrame,
489                             T2CA(frame->m_strName), NULL, 0, NULL, &frame_node));
490             }
491
492         CComPtr<ID3DXFileSaveData> matrix_node;
493         THR(frame_node->AddDataObject(TID_D3DRMFrameTransformMatrix,
494             NULL, NULL, sizeof(D3DMATRIX), &frame->m_mat,
495             &matrix_node));
496
497         if (frame->m_pMesh)
498         {
499             save_mesh(so, frame_node, frame->m_pMesh);
500         }
501
502         if (frame->m_pChild)
503         {
504             save(so, frame_node, frame->m_pChild);
```

```
505       }
506
507       if (frame->m_pNext)
508       {
509           save(so, parent, frame->m_pNext);
510       }
511
512       if (!parent)
513       {
514           THR(so->Save());
515           s_buffer.clear();
516           s_strings.clear();
517       }
518   }
519
520   ////////////////////////////////////////////////////////////////
521   // c_frame::save
522   //
523   // Create a save object for the given file and save this frame
524   // to the file.
525   //
526   void
527   c_frame::save(const rt::tstring &filename,
528                 bool binary, bool compressed)
529   {
530       CComPtr<ID3DXFileSaveObject> so;
531       {
532           CComPtr<ID3DXFile> xfile;
533           THR(::D3DXFileCreate(&xfile));
534           THR(xfile->RegisterTemplates(rt::D3DRM_XTEMPLATES,
535               rt::D3DRM_XTEMPLATE_BYTES));
536
537           USES_CONVERSION;
538           THR(xfile->CreateSaveObject(T2CA(filename.c_str()),
539                           D3DXF_FILESAVE_TOFILE,
540               (compressed ? D3DXF_FILEFORMAT_COMPRESSED : 0) |
541               (binary ? D3DXF_FILEFORMAT_BINARY : D3DXF_FILEFORMAT_TEXT),
542               &so));
543       }
544       save(so, NULL, this);
545   }
```