Chapter 15

D3DX Utility Library

"Nothing is ever said that has not been said before."
Terence: *Eunuchus*, prologue, c. 160 B.C.

15.1 Overview

D3DX is a static library designed for production use in shipping applications. D3DX provides operations that are common in 3D graphics applications, but are not necessarily tied directly to the graphics pipeline. D3DX is a static library, so you do not have any runtime dependencies when you use it. D3DX consists of several areas of related operations: abstract data types, helper objects, triangle meshes, resource functions, and miscellaneous functions.

The concrete classes providing data types for colors, materials, vectors, matrices, planes and quaternions. They are implemented as C++ classes and associated global functions operating on those classes. The data types are summarized in section 15.3 and described in detail in chapter 16.

The helper objects implement matrix stacks, text rendering, sprites, multipass rendering effects, and render target assistance. These objects are implemented as COM objects. They are summarized in section 15.4 and described in detail in chapter 17.

Most of the higher level operations in D3DX operate on triangle meshes. Meshes come in several varieties, all implemented as COM objects. Mesh simplification is exposed through several COM interfaces, providing dynamic level of detail selection. Skinned meshes are also provided as a COM object. Functions are provided for operating on meshes and streaming them to and from permanent storage. Meshes are summarized in section 15.5 and described in detail in chapter 19.

The resource functions in D3DX handle some of the details of creating, initializing, and streaming resources to and from files within the requirements of a device. The resource funcions are described in section 15.6.

D3DX also provides a few bits and pieces not easily categorized with the portions described so far. They are described in section 15.16

15.2 Headers and Libraries

D3DX is provided as a collection of header files and a library. The library comes in both debug and release versions. For debugging, a DLL library is also provided, but this cannot be used with shipping applications.

To compile against D3DX, you should include the header file `<d3dx9.h>`. This header file includes the remaining D3DX headers: `<d3dx9core.h>`, `<d3dx9effect.h>`, `<d3dx9math.h>`, `<d3dx9mesh.h>`, `<d3dx9shape.h>`, and `<d3dx9tex.h>`.

The declarations declared by these header files are defined in library files. The library file `d3dx9.lib` is a static library to be used in release builds of your application. It contains optimized versions of the D3DX library routines and interfaces and is compiled without debug diagnostic information. The library file `d3dx9dt.lib` contains debug versions of the D3DX library routines and interfaces and is compiled with additional debugging diagnostic information. The library file `d3dx9d.lib` contains an import library for the debug DLL `d3dx9d.dll` and should only be used for debugging builds. The DLL will only be present on a machine where the DirectX SDK has been installed.

15.3 Data Types

D3DX provides some **concrete datatypes** and functions for manipulating them. A concrete datatype exposes the programmer to the details of the type's members and implementation, while an abstract datatype insulates the programmer from the details of the implementation. D3DX exposes colors, vectors, planes, quaternions and matrices as concrete datatypes.

The `D3DXCOLOR` class encapsulated four floating-point values representing an RGBA color. Vector operations are provided through the `D3DXVECTOR2`, `D3DXVECTOR3`, and `D3DXVECTOR4` classes. The `D3DXPLANE` class provides a three-dimensional plane datatype. The `D3DXQUATERNION` class provides a quaternion datatype, useful for construction orienting transformations. The `D3DXMATRIX` class encapsulates 4×4 homogeneous transformation matrices.

The details of these concrete datatypes and their associated functions are described in chapter 16.

15.4 Helper Objects

In contrast to the concrete datatypes that are exposed as C++ classes, D3DX also provides some abstract datatypes implemented as COM objects. The usual reference counting rules for COM objects must be observed when using these helper objects, unlike the concrete data types.

`ID3DXMatrixStack` provides a stack of transformation matrices for use in scene graph traversal. `ID3DXFont` provides text rendering implemented using GDI into a memory DC. `ID3DXSprite` provides simple 2D oriented sprites implemented using textured quads. `ID3DXRenderToSurface` provides a simplified interface for rendering to a target surface. Similarly, `ID3DXRenderToEnvMap` provides a simplified interface for rendering to environment map textures.

`ID3DXEffect` encapsulates a rendering "effect", which is implemented using one or more rendering "techniques". Each technique encapsulates a group of associated rendering state and rendering passes necessary to achieve the desired effect. Effects support parameters to the rendering techniques to encapsulate variable state that may change with the portion of the scene to be rendered with the effect.

The details of these COM helper objects and their associated functions are described in chapter 17.

15.5 Meshes

D3DX provides several mesh objects that encapsulate indexed triangle lists. Mesh simplification and progressive mesh support are provided. `ID3DXMesh` interface encapsulates a shape as a triangle mesh. Progressive mesh refinement is available in the `ID3DXPMesh` interface. `ID3DXBaseMesh` is a base interface used by the `ID3DXMesh` and `ID3DXPMesh` objects. Arbitrary mesh simplification is encapsulated by `ID3DXSPMesh`. Finally, skinned meshes – meshes using vertex blending – are encapsulated by the `ID3DXSkinInfo`. D3DX also provides support for reading and writing mesh objects to and from `.x` files.

The details of the mesh objects and their associated functions are described in chapter 19.

15.6 Resources

Images and textures are resources that are used by almost every Direct3D application. D3DX provides a wide variety of functions for creating and initializing surfaces and textures from files, Win32 resources, and raw memory blocks. D3DX can read `.bmp`, `.dds`, `.dib`, `.jpg`, `.png`, `.ppm`, and `.tga` file formats when reading resource data. D3DX can write resource data to DirectDraw surface (`.dds`) files and Windows bitmap format (`.bmp`) files.

File data on disk is referenced by its file name. A file in memory is referenced by a pointer to a block of memory containing the file data and the size of the memory block. A file in a Win32 resource is referenced by a handle to the module containing the resource and the name of the resource. Win32 resources can be of type `RT_BITMAP` for `.bmp` data or `RT_RCDATA` for other file formats. Be careful not to confuse Win32 resources, data stored in an executable file such as a `.dll` or `.exe` file, with Direct3D resources, such as textures, image surfaces, vertex buffers and index buffers.

Many of the D3DX functions that manipulate resources take parameters controlling behavior that accept a value of D3DX_DEFAULT. D3DX_DEFAULT is a non-zero value indicating that a default behavior for the parameter should be used instead of a specific behavior indicated by a specific parameter value. The relevant default behavior is discussed where the parameter is discussed.

```
#define D3DX_DEFAULT ULONG_MAX
```

15.7 Resource Image Information

The simplest operation supported by D3DX is to read the dimension and format information associated with an image file. The functions D3DXGetImageInfo-FromFile, D3DXGetImageInfoFromFileInMemory, and D3DXGetImageInfoFrom-Resource retrieve the image information from a file on disk, a file located in memory, or a file stored as a Win32 resource.

```
HRESULT ::D3DXGetImageInfoFromFile(LPCTSTR filename,
            D3DXIMAGE_INFO *result);

HRESULT ::D3DXGetImageInfoFromFileInMemory(const void *data,
            UINT size,
            D3DXIMAGE_INFO *result);

HRESULT ::D3DXGetImageInfoFromResource(HMODULE module,
            LPCTSTR resource,
            D3DXIMAGE_INFO *result);

typedef struct _D3DXIMAGE_INFO
{
    UINT                 Width;
    UINT                 Height;
    UINT                 Depth;
    UINT                 MipLevels;
    D3DFORMAT            Format;
    D3DRESOURCETYPE      ResourceType;
    D3DXIMAGE_FILEFORMAT ImageFileFormat;
} D3DXIMAGE_INFO;
```

The file is described by a D3DXIMAGEINFO structure. The structure contains the pixel dimensions of the image or volume. For images, the Depth member is set to 1. Most images will have a value of 1 for the MipLevels member, but .dds files can store entire mipmap chains. The ResourceType member indicates if the file contains a texture, volume texture or cube texture. The Format member gives the pixel format that most closely matches the data in the file. The ImageFileFormat member gives the file format of the resource data and is one of the values in the D3DXIMAGE_FILEFORMAT enumeration.

```
typedef enum _D3DXIMAGE_FILEFORMAT
{
    D3DXIFF_BMP = 0,
    D3DXIFF_DDS = 4,
    D3DXIFF_DIB = 6,
    D3DXIFF_JPG = 1,
    D3DXIFF_PNG = 3,
    D3DXIFF_PPM = 5,
    D3DXIFF_TGA = 2
} D3DXIMAGE_FILEFORMAT;
```

15.8 Resource Requirements

A common task in loading resources from files is to match the pixel dimensions of the file to the requirements of the device. D3DX provides the functions `D3DXCheckTextureRequirements`, `D3DXCheckCubeTextureRequirements` and `D3DXCheckVolumeTextureRequirements` for adjusing resource dimensions, number of mipmap levels, and pixel format to fit within the requirements of the device.

`CubeTexture`—textbf`VolumeTexture`—textbf

```
HRESULT ::D3DXCheckTextureRequirements(
            IDirect3DDevice9 *device,
            UINT *width,
            UINT *height,
            UINT *mip_levels,
            DWORD usage,
            D3DFORMAT *format,
            D3DPOOL pool);

HRESULT ::D3DXCheckCubeTextureRequirements(
            IDirect3DDevice9 *device,
            UINT *size,
            UINT *mip_levels,
            DWORD usage,
            D3DFORMAT *format,
            D3DPOOL pool);

HRESULT ::D3DXCheckVolumeTextureRequirements(
            IDirect3DDevice9 *device,
            UINT *width,
            UINT *height,
            UINT *depth,
            UINT *mip_levels,
            DWORD usage,
            D3DFORMAT *format,
            D3DPOOL pool);
```

These functions use the device capabilities and the `CheckDeviceFormat` method of `IDirect3D9` to adjust the texture creation parameters for the device. Upon succesful return, the returned parameters can be used to create a valid texture resource on the device. The `usage` and `pool` arguments are used to validate the requested resource and are not adjusted by the function.

15.9 Format Conversion

D3DX provides routines for converting pixel data from one format to another. This may involve color space conversion, channel rescaling, compression or decompression, and filtering depending on the source and destination pixel formats. `D3DXLoadSurfaceFromSurface` performs format conversion on surfaces and `D3DXLoadVolumeFromVolume` performs format conversion on volumes. To perform conversions between textures, use the `GetSurfaceLevel` or `GetVolume-Level` methods to obtain the necessary interface pointers for each contained surface or volume within the textures.

```
HRESULT ::D3DXLoadSurfaceFromSurface(
            IDirect3DSurface9 *destination,
            const PALETTEENTRY *dest_palette,
            const RECT *dest_rect,
            IDirect3DSurface9 *source,
            const PALETTEENTRY *source_palette,
            const RECT *source_rect,
            DWORD filter,
            D3DCOLOR color_key);

HRESULT ::D3DXLoadVolumeFromVolume(
            IDirect3DVolume9 *destination,
            const PALETTEENTRY *dest_palette,
            const D3DBOX *dest_box,
            IDirect3DVolume9 *source,
            const PALETTEENTRY *source_palette,
            const D3DBOX *source_box,
            DWORD filter,
            D3DCOLOR color_key);
```

The `dest_palette` and `source_palette` arguments are pointers to a block of memory containing 256 `PALETTEENTRY` structures when the source or destination are in `D3DFMT_P8` format. These arguments may be `NULL` when the source or destination is not an indexed format.

The `dest_rect` and `source_rect` arguments allow the conversion operation to be applied to subrectangles of the source and destination surfaces. The `dest_box` and `source_box` arguments operate similarly for volumes. When these arguments refer to regions of different size in the source and destination, D3DX performs filtering on the regions to resize the source to fit the destination. The filter kernel used is specified by the `filter` parameter, whose value can be `D3DX_DEFAULT` or one or more of the following flags:

`D3DX_FILTER`—textbf

```
#define D3DX_FILTER_NONE     (1 << 0)
#define D3DX_FILTER_POINT    (2 << 0)
#define D3DX_FILTER_LINEAR   (3 << 0)
#define D3DX_FILTER_TRIANGLE (4 << 0)
#define D3DX_FILTER_BOX      (5 << 0)

#define D3DX_FILTER_MIRROR_U (1 << 16)
#define D3DX_FILTER_MIRROR_V (2 << 16)
#define D3DX_FILTER_MIRROR_W (4 << 16)
#define D3DX_FILTER_MIRROR   (7 << 16)
#define D3DX_FILTER_DITHER   (8 << 16)
```

The value `D3DX_FILTER_NONE` results in pixels in the destination region that lie outside the source region to be replaced with transparent black. With this value, no scaling or filtering will be performed. The value `D3DX_DEFAULT` is equivalent to `D3DX_FILTER_TRIANGLE` combined with `D3DX_FILTER_DITHER`.

The values `D3DX_FILTER_POINT`, `D3DX_FILTER_LINEAR`, `D3DX_FILTER_TRIANGLE` or `D3DX_FILTER_BOX` specify the filter kernel to be applied to the source region. Point filtering is the fastest rescaling method, but also introduces the most aliasing. Linear filtering uses a 2×2 pixel neighbourhood for a surface, or a $2 \times 2 \times 2$ pixel neighbourhood for a volume, and linearly interpolates between the source pixels in the neighbourhood to compute the destination pixel value. Linear filtering is more expensive than point filtering but results in a higher quality image. The triangle filter uses a neighbourhood proportional to the relative size of the source and destination areas, making this the most expensive filter with the highest quality. The box filter uses a 2×2 surface neighbourhood, or $2 \times 2 \times 2$ for volumes, and computes the destination pixel value as the average of all pixels within the source neighbourhood. The box filter is typically used for computing mipmap levels, but still suffers from some aliasing artifacts.

The mirror flags control repetition of the source image with respect to the x, y, or z axes of the source image. The `D3DX_FILTER_MIRROR` flag is a shorthand for mirroring about all axes of the source image. The `D3DX_FILTER_DITHER` flag enables a 4×4 ordered dither of the source pixels to the destination pixels. Dithering is most useful in preventing banding artifacts when reducing the color depth of the source.

Finally, the `color_key` argument allows the application to specify a pixel value in the source that will be substituted with transparent black during processing. While Direct3D itself does not support color keying transparency, D3DX provides this limited form of color keying during surface loading.

15.10 Creating Texture Resources

D3DX provides two forms of functions for creating texture resources from files on disk, files in memory, or files in Win32 resources. The first form uses the resource checking and filtering functions just described to construct a resource that "just works", possibly resizing the resource and converting the pixel format to the requirements of the device. The second form is an extended version that allows the application to finely control the process of creating the texture resource.

15.10.1 Creating Uninitialized Textures

The functions `D3DXCreateTexture`, `D3DXCreateCubeTexture`, and `D3DXCreate-VolumeTexture` create texture resources whose contents are uninitialized. You can initialize the texture contents with some application code after the resource has been created. These functions call the corresponding texture resource requirements functions to adjust the input parameters to suit the device and then call the corresponding method on the device to create the appropriate resource. The resulting interface pointer is returned to the caller.

```
HRESULT ::D3DXCreateTexture(IDirect3DDevice9 *device,
            UINT width,
            UINT height,
            UINT mip_levels,
            DWORD usage,
            D3DFORMAT format,
            D3DPOOL pool,
            IDirect3DTexture9 **result);

HRESULT ::D3DXCreateCubeTexture(IDirect3DDevice9 *device,
            UINT Size,
            UINT mip_levels,
            DWORD usage,
            D3DFORMAT format,
            D3DPOOL pool,
            IDirect3DCubeTexture9 **result);

HRESULT ::D3DXCreateVolumeTexture(IDirect3DDevice9 *device,
            UINT width,
            UINT height,
            UINT depth,
            UINT mip_levels,
            DWORD usage,
            D3DFORMAT format,
            D3DPOOL pool,
            IDirect3DVolumeTexture9 **result);
```

15.10.2 Creating Textures From Files

The functions D3DXCreateCubeTextureFromFile, D3DXCreateTextureFromFile, and D3DXCreateVolumeTextureFromFile create texture resources from files on disk. These functions obtain the pixel dimensions and pixel format from the data stored in the file and adjust the dimensions and format to meet the restrictions of the device for the resource. The pixel data in the file is read into memory, and possibly converted to the pixel format used by the resource, before being copied into the resource. These functions provide a very easy way to initialize a texture resource from an image file.

```
HRESULT ::D3DXCreateTextureFromFile(
            IDirect3DDevice9 *device,
            LPCTSTR filename,
            IDirect3DTexture9 **result);

HRESULT ::D3DXCreateCubeTextureFromFile(
            IDirect3DDevice9 *device,
            LPCTSTR filename,
            IDirect3DCubeTexture9 **result);%

HRESULT ::D3DXCreateVolumeTextureFromFile(
            IDirect3DDevice9 *device,
            LPCTSTR filename,
            IDirect3DVolumeTexture9 **result);
```

Texture resources can be created from any of the supported file formats enumerated in D3DXIMAGE_FILEFORMAT. Cube textures can only be created from .dds files. Volume textures can be created from .dds files which can contain an entire volume, or from image files to create a volume containing a single slice.

The extended texture creation functions allow all the parameters of the texture creation process to be controlled by the caller. The caller can specify target resource properties and file processing arguments to obtain fine control over the resource creation process. The resource dimensions, usage, format, and memory pool can all be specified individually, or left to default arguements with D3DX_DEFAULT. Texture filtering parameters and a color to be treated as a transparent color key during the loading process can be specified.

```
HRESULT ::D3DXCreateTextureFromFileEx(
            IDirect3DDevice9 *device,
            LPCTSTR filename,
            UINT width,
            UINT height,
            UINT mip_levels,
            DWORD usage,
            D3DFORMAT format,
            D3DPOOL pool,
            DWORD filter,
            DWORD mip_filter,
            D3DCOLOR color_key,
            D3DXIMAGE_INFO *info,
            PALETTEENTRY *palette,
            IDirect3DTexture9 **result);

HRESULT ::D3DXCreateCubeTextureFromFileEx(
            IDirect3DDevice9 *device,
            LPCTSTR filename,
            UINT Size,
            UINT mip_levels,
            DWORD usage,
            D3DFORMAT format,
            D3DPOOL pool,
            DWORD filter,
            DWORD mip_filter,
            D3DCOLOR color_key,
            D3DXIMAGE_INFO *info,
            PALETTEENTRY *palette,
            IDirect3DCubeTexture9 **result);

HRESULT ::D3DXCreateVolumeTextureFromFileEx(
            IDirect3DDevice9 *device,
            LPCTSTR filename,
            UINT width,
            UINT height,
            UINT depth,
            UINT mip_levels,
            DWORD usage,
            D3DFORMAT format,
            D3DPOOL pool,
            DWORD filter,
            DWORD mip_filter,
            D3DCOLOR color_key,
            D3DXIMAGE_INFO *info,
            PALETTEENTRY *palette,
            IDirect3DVolumeTexture9 **result);
```

The `filter` parameter controls the filtering of the image into the most detailed level of the texture resource while the `mip_filter` argument controls the creation of mipmap levels. When `D3DX_DEFAULT` is used for the `filter` parameter, a triangle filter with dithering is used to resample the image in the file. When `D3DX_DEFAULT` is used for the `mip_filter`, a box filter is used to generate the mipmap levels. The `info` parameter returns information about the image file that was used to construct the texture. If this information is not needed, an application can pass NULL for the info parameter. The palette parameter is used to return the palette loaded with the resource. This parameter can be NULL if no palette is needed.

15.10.3 Creating Textures From Files In Memory

If the file is located in a block of memory, it can be used to create a resource with D3DXCreateTextureFromFileInMemory, D3DXCreateCubeTextureFromFileInMemory and D3DXCreateVolumeTextureFromFileInMemory. The Ex forms are similar to the extended texture creation functions discussed earlier in this section.

```
HRESULT ::D3DXCreateTextureFromFileInMemory(
            IDirect3DDevice9 *device,
            const void *data,
            UINT size,
            IDirect3DTexture9 **result);

HRESULT ::D3DXCreateCubeTextureFromFileInMemory(
            IDirect3DDevice9 *device,
            const void *data,
            UINT size,
            IDirect3DCubeTexture9 **result);

HRESULT ::D3DXCreateVolumeTextureFromFileInMemory(
            IDirect3DDevice9 *device,
            const void *data,
            UINT size,
            IDirect3DVolumeTexture9 **result);

HRESULT ::D3DXCreateTextureFromFileInMemoryEx(
            IDirect3DDevice9 *device,
            const void *data,
            UINT size,
            UINT width,
            UINT height,
            UINT mip_levels,
            DWORD usage,
            D3DFORMAT format,
            D3DPOOL pool,
            DWORD filter,
            DWORD mip_filter,
            D3DCOLOR color_key,
            D3DXIMAGE_INFO *info,
            PALETTEENTRY *palette,
            IDirect3DTexture9 **result);
```

```
HRESULT ::D3DXCreateCubeTextureFromFileInMemoryEx(
            IDirect3DDevice9 *device,
            const void *data,
            UINT size,
            UINT Size,
            UINT mip_levels,
            DWORD usage,
            D3DFORMAT format,
            D3DPOOL pool,
            DWORD filter,
            DWORD mip_filter,
            D3DCOLOR color_key,
            D3DXIMAGE_INFO *info,
            PALETTEENTRY *palette,
            IDirect3DCubeTexture9 **result);

HRESULT ::D3DXCreateVolumeTextureFromFileInMemoryEx(
            IDirect3DDevice9 *device,
            const void *data,
            UINT size,
            UINT width,
            UINT height,
            UINT depth,
            UINT mip_levels,
            DWORD usage,
            D3DFORMAT format,
            D3DPOOL pool,
            DWORD filter,
            DWORD mip_filter,
            D3DCOLOR color_key,
            D3DXIMAGE_INFO *info,
            PALETTEENTRY *palette,
            IDirect3DVolumeTexture9 **result);
```

15.10.4 Creating Textures From Resources

Creating a texture from a file stored as a Win32 resource is very similar to creating a texture from a file stored on disk or stored in memory. The only difference is how the source of the texture data is specified. To identify a file stored in a resource you supply the handle to the module containing the resource and the name of the resource.

The functions `D3DXCreateTextureFromResource`, `D3DXCreateCubeTexture-FromResource`, and `D3DXCreateVolumeTextureFromResource` are used to create texture resources on a device from file data stored in Win32 resources associated with an executable module. The extended forms allow the usual control over the resource creation parameters.

```
HRESULT ::D3DXCreateTextureFromResource(
    IDirect3DDevice9 *device,
    HMODULE module,
    LPCTSTR resource,
    IDirect3DTexture9 **result);

HRESULT ::D3DXCreateCubeTextureFromResource(
    IDirect3DDevice9 *device,
    HMODULE module,
    LPCTSTR resource,
    IDirect3DCubeTexture9 **result);

HRESULT ::D3DXCreateVolumeTextureFromResource(
    IDirect3DDevice9 *device,
    HMODULE module,
    LPCTSTR resource,
    IDirect3DVolumeTexture9 **result);

HRESULT ::D3DXCreateTextureFromResourceEx(
    IDirect3DDevice9 *device,
    HMODULE module,
    LPCTSTR resource,
    UINT width,
    UINT height,
    UINT mip_levels,
    DWORD usage,
    D3DFORMAT format,
    D3DPOOL pool,
    DWORD filter,
    DWORD mip_filter,
    D3DCOLOR color_key,
    D3DXIMAGE_INFO *info,
    PALETTEENTRY *palette,
    IDirect3DTexture9 **result);

HRESULT ::D3DXCreateCubeTextureFromResourceEx(
    IDirect3DDevice9 *device,
    HMODULE module,
    LPCTSTR resource,
    UINT Size,
    UINT mip_levels,
    DWORD usage,
    D3DFORMAT format,
    D3DPOOL pool,
    DWORD filter,
    DWORD mip_filter,
    D3DCOLOR color_key,
    D3DXIMAGE_INFO *info,
    PALETTEENTRY *palette,
    IDirect3DCubeTexture9 **result);

HRESULT ::D3DXCreateVolumeTextureFromResourceEx(
    IDirect3DDevice9 *device,
    HMODULE module,
    LPCTSTR resource,
    UINT width,
    UINT height,
    UINT depth,
    UINT mip_levels,
    DWORD usage,
    D3DFORMAT format,
    D3DPOOL pool,
    DWORD filter,
    DWORD mip_filter,
    D3DCOLOR color_key,
    D3DXIMAGE_INFO *info,
    PALETTEENTRY *palette,
    IDirect3DVolumeTexture9 **result);
```

For example, if the file "`foo.png`" is stored in a data resource with the integer name `IDR_DATA1` in the module associated with the process' executable image, you can create a texture from that resource with the following code snippet:

```
IDirect3DTexture9 *texture = 0;
::D3DXCreateTextureFromResource(device, NULL,
    MAKEINTRESOURCE(IDR_DATA1), &texture);
```

15.11 Loading Resources

If you have an existing surface or texture resource and want to load images into that resource, you can use D3DX read the contents into the surface. For a texture or cube texture resource, you can obtain the surface interface pointer for a particular level and then load the data into the surface. For a volume texture you can obtain the volume interface pointer for a particular level and then load the data into the volume.

The functions `D3DXLoadSurfaceFromMemory`, `D3DXLoadSurfaceFromFile`, `D3DXLoadSurfaceFromFileInMemory`, and `D3DXLoadSurfaceFromResource` load data into surfaces. The functions `D3DXLoadVolumeFromMemory`, `D3DXLoadVolume-FromFile`, `D3DXLoadVolumeFromFileInMemory` and `D3DXLoadVolumeFromResource` load data into volumes.

The `FromMemory` versions of the functions load a surface or volume from a region of memory containing raw pixel data. The raw pixel data is consistent with the `format` and `pitch` parameters. Surface data is arranged as a sequence of scanlines, beginning with the bottom-most scanline in the data, with each scanline offset from the previous scanline by `pitch` bytes in memory. Volume data is arranged as a sequence of slices, beginning with the bottom-most slice. Each slice is arranged as a surface. The `slice_pitch` and `row_pitch` arguments give the distance in bytes between successive slices within the volume and rows within a slice, respectively.

The remaining arguments are identical to those described earlier in this chapter.

```
HRESULT ::D3DXLoadSurfaceFromMemory(
    IDirect3DSurface9 *destination,
    const PALETTEENTRY *dest_palette,
    const RECT *dest_rect,
    const void *source,
    D3DFORMAT format,
    UINT pitch,
    const PALETTEENTRY *source_palette,
    const RECT *source_rect,
    DWORD filter,
    D3DCOLOR color_key);

HRESULT ::D3DXLoadSurfaceFromFile(
    IDirect3DSurface9 *destination,
    const PALETTEENTRY *dest_palette,
    const RECT *dest_rect,
    LPCTSTR filename,
    const RECT *source_rect,
    DWORD filter,
    D3DCOLOR color_key,
    D3DXIMAGE_INFO *info);

HRESULT ::D3DXLoadSurfaceFromFileInMemory(
    IDirect3DSurface9 *destination,
    const PALETTEENTRY *dest_palette,
    const RECT *dest_rect,
    const void *data,
    UINT size,
    const RECT *source_rect,
    DWORD filter,
    D3DCOLOR color_key,
    D3DXIMAGE_INFO *info);

HRESULT ::D3DXLoadSurfaceFromResource(
    IDirect3DSurface9 *destination,
    const PALETTEENTRY *dest_palette,
    const RECT *dest_rect,
    HMODULE module,
    LPCTSTR resource,
    const RECT *source_rect,
    DWORD filter,
    D3DCOLOR color_key,
    D3DXIMAGE_INFO *info);

HRESULT ::D3DXLoadVolumeFromMemory(
    IDirect3DVolume9 *destination,
    const PALETTEENTRY *dest_palette,
    const D3DBOX *dest_box,
    const void *source,
    D3DFORMAT format,
    UINT row_pitch,
    UINT slice_pitch,
    const PALETTEENTRY *source_palette,
    const D3DBOX *source_box,
    DWORD filter,
    D3DCOLOR color_key);

HRESULT ::D3DXLoadVolumeFromFile(
    IDirect3DVolume9 *destination,
    const PALETTEENTRY *dest_palette,
    const D3DBOX *dest_box,
    LPCTSTR filename,
    const D3DBOX *source_box,
    DWORD filter,
    D3DCOLOR color_key,
    D3DXIMAGE_INFO *info);

HRESULT ::D3DXLoadVolumeFromFileInMemory(
    IDirect3DVolume9 *destination,
    const PALETTEENTRY *dest_palette,
    const D3DBOX *dest_box,
    const void *data,
    UINT size,
    const D3DBOX *source_box,
    DWORD filter,
    D3DCOLOR color_key,
    D3DXIMAGE_INFO *info);

HRESULT ::D3DXLoadVolumeFromResource(
    IDirect3DVolume9 *destination,
    const PALETTEENTRY *dest_palette,
    const D3DBOX *dest_box,
    HMODULE module,
    LPCTSTR resource,
    const D3DBOX *source_box,
    DWORD filter,
    D3DCOLOR color_key,
    D3DXIMAGE_INFO *info);
```

15.12 Saving Resources

D3DX can save surface, texture and volume texture resource data to the `.bmp` and `.dds` file formats. The bitmap file format is supported for images, single-level textures and single-slice volumes. The DirectDraw surface format is supported for all resource types. A portion of the resource can be saved to the file using the `rect` and `box` parameters. If the resource is in a palette-based format, the `palette` parameter supplies the color palette to be saved in the file.

```
HRESULT ::D3DXSaveSurfaceToFile(LPCTSTR filename,
    D3DXIMAGE_FILEFORMAT file_format,
    IDirect3DSurface9 *source,
    const PALETTEENTRY *palette,
    const RECT *rect);

HRESULT ::D3DXSaveTextureToFile(LPCTSTR filename,
    D3DXIMAGE_FILEFORMAT file_format,
    IDirect3DBaseTexture9 *source,
    const PALETTEENTRY *palette);

HRESULT ::D3DXSaveVolumeToFile(LPCTSTR filename,
    D3DXIMAGE_FILEFORMAT file_format,
    IDirect3DVolume9 *source,
    const PALETTEENTRY *palette,
    const D3DBOX *box);
```

15.13 Filling Textures

In addition to loading resources with images and volumes from files, D3DX provides a way for you to procedurally fill a texture resource.[1] The functions `D3DX-FillTexture`, `D3DXFillCubeTexture` and `D3DXFillVolumeTexture` each take a pointer to a function supplied by the application in the `fill_proc` parameter. D3DX calls this function for each pixel in the resource with the coordinates of the pixel to be filled and the `context` argument passed to the fill function.

[1]There is no function provided to procedurally fill a surface.

The `context` argument allows the fill function to obtain information from the application.

```
HRESULT ::D3DXFillTexture(IDirect3DTexture9 *texture,
    LPD3DXFILL2D fill_proc,
    void *context);

HRESULT ::D3DXFillCubeTexture(
    IDirect3DCubeTexture9 *texture,
    LPD3DXFILL3D fill_proc,
    void *context);

HRESULT ::D3DXFillVolumeTexture(
    IDirect3DVolumeTexture9 *texture,
    LPD3DXFILL3D fill_proc,
    void *context);
```

`D3DXFILL2D` is the fill procedure used for filling texture resources. Cube textures and volume textures are filled with a three dimensional fill procedure `D3DXFILL3D`.

```
typedef void (*LPD3DXFILL2D)(D3DXVECTOR4 *result,
            D3DXVECTOR2 *coord,
            D3DXVECTOR2 *size,
            void *context);

typedef void (*LPD3DXFILL3D)(D3DXVECTOR4 *result,
            D3DXVECTOR3 *coord,
            D3DXVECTOR3 *size,
            void *context);
```

The fill procedure receives the texture coordinates in the `coord` vector. The coordinates will be drawn from $[0, 1]$ for textures and volume textures, while cube texture coordinates are drawn from $[-1, 1]$. For a cube texture, one of the three coordinates will always have the value 1 or -1 indicating which face of the cube is being filled. The `size` vector describes the region of the texture being sampled, with larger values corresponding to larger regions of texture space.

15.14 Filtering Textures

When an application changes the most detailed level of a mipmapped texture, the less detailed levels of the mipmap chain need to be regenerated. D3DX provides a single function to create the mipmap levels by filtering. Since all texture resources derive from `IDirect3DBaseTexture9`, only a single function is needed. The function determines the texture type and filters the texture level given by the `level` parameter to produce the lesser detailed levels. If `D3DX_-DEFAULT` is used for the `level` parameter, then level 0 is used to construct all the other mipmap levels. If the `filter` parameter is `D3DX_DEFAULT`, then a box filter is used.

```
HRESULT ::D3DXFilterTexture(
    IDirect3DBaseTexture9 *texture,
    const PALETTEENTRY *palette,
    UINT level,
    DWORD filter);
```

15.15 Creating Normal Maps

D3DX can create a normal map from a height field. The `destination` argument gives the texture into which the normal map will be stored. The destination texture must already exist, it is not created by this function. The `source` argument gives the texture in which the height field is stored. Each pixel in the height field is treated as an elevation and a 3×3 neighbourhood around the source pixel is used to compute the normal to the height field.

```
HRESULT ::D3DXComputeNormalMap(IDirect3DTexture9 *destination,
    IDirect3DTexture9 *source,
    const PALETTEENTRY *palette,
    DWORD flags,
    DWORD channel,
    float amplitude);
```

The `channel` argument specifies which channel from the height field is used as the elevation. It can be one of the following values.

```
#define D3DX_CHANNEL_RED       (1 << 0)
#define D3DX_CHANNEL_BLUE      (1 << 1)
#define D3DX_CHANNEL_GREEN     (1 << 2)
#define D3DX_CHANNEL_ALPHA     (1 << 3)
#define D3DX_CHANNEL_LUMINANCE (1 << 4)
```

The `flags` argument supplies options to the computation of the normal map and is a combination of zero or more flags. The mirroring flags control the repetition of the height field in the x and y directions, with `D3DX_NORMALMAP_MIRRORU` and `D3DX_NORMALMAP_MIRRORV`, respectively. The `D3DX_NORMALMAP_MIRROR` flag enables mirroring in both directions. The `D3DX_NORMALMAP_INVERTSIGN` flag reverses the direction of the computed normal. The `D3DX_NORMALMAP_COMPUTEOCCLUSION` flag computes a per-pixel occlusion value and stores it in the alpha channel of the resulting normal map. A value of 0 means that the pixel is completely obscured and a value of 1 means that the pixel is not obscured in any direction.

```
#define D3DX_NORMALMAP_MIRROR_U          (1 << 16)
```

```
#define D3DX_NORMALMAP_MIRROR_V          (2 << 16)
#define D3DX_NORMALMAP_MIRROR            (3 << 16)
#define D3DX_NORMALMAP_INVERTSIGN        (8 << 16)
#define D3DX_NORMALMAP_COMPUTE_OCCLUSION (16 << 16)
```

15.16 Miscellaneous

Besides data types, helper objects, triangle meshes, and resource functions, D3DX provides some other operations that don't fit into any of the other categories.

Mathematical constant macros, unit conversion macros

15.16.1 Macros

D3DX supplies some trigonometric macros. The macros `D3DX_PI` and `D3DX_1BYPI` define constants for the value of π and $1/\pi$. The macros `D3DXToDegree` and `D3DXToRadian` provide conversions between degrees and radians.

```
const float D3DX_PI;
const float D3DX_1BYPI;

float D3DXToRadian(float degrees);
float D3DXToDegree(float radians);
```

15.16.2 Fresnel's Formulas

Fresnel's formulas describe the amount of light reflected at an interface between two materials. The formulas give the amount of reflected light as a sum of two polarized components: light polarized parallel ($F_\parallel$) to the interface surface and light polarized perpendicular ($F_\perp$) to the interface surface. The amount of reflected light depends on the angle θ of the incident light, the index of refraction of the materials N, the geometry of the interface and the light as well as the wavelength of the light.

A material is characterized by its complex index of refraction $N = \eta + i\kappa$, where η is the simple index of refraction for the material and κ is the material's extinction coefficient. When considering reflection at the interface between the two materials, the relative index of refraction may be used:

$$N = \frac{N_1}{N_2} = \frac{\eta_1 + i\kappa_1}{\eta_2 + i\kappa_2} = \frac{\eta_1\eta_2 + \kappa_1\kappa_2}{\eta_1^2 + \kappa_1^2} + i\frac{\eta_1\eta_2 - \kappa_1\kappa_2}{\eta_1^2 + \kappa_1^2}$$

The parallel component of the reflected light, $F_\parallel$, and the perpendicular component of the reflected light, $F_\perp$, are given by Fresnel's formulas as follows, using the relative index of refraction N and the angle of the incident light θ.

$$\begin{aligned}
F_\perp &= \frac{a^2 + b^2 - 2a\cos\theta + \cos^2\theta}{a^2 + b^2 + 2a\cos\theta + \cos^2\theta} \\
F_\parallel &= F_\perp \frac{a^2 + b^2 - 2a\sin\theta\tan\theta + \sin^2\theta\tan^2\theta}{a^2 + b^2 + 2a\sin\theta\tan\theta + \sin^2\theta\tan^2\theta} \\
a &= \sqrt{\frac{1}{2}(c + d)} \\
b &= \sqrt{\frac{1}{2}(c - d)} \\
c &= \sqrt{d^2 + 4\eta^2\kappa^2} \\
d &= \eta^2 - \kappa^2 - \sin^2\theta
\end{aligned}$$

For an interface between two simple materials, the simple relative index of refraction is the ratio of the index of fraction of the two materials, η_1/η_2. For instance, the relative index of refraction between air and glass is about 1.5. In this simplified case, $\kappa_1 = \kappa_2 = 0$ and the reflectance terms are simplified to the following formulas.

$$\begin{aligned}
F_\perp &= \frac{a^2 - 2a\cos\theta + \cos^2\theta}{a^2 + 2a\cos\theta + \cos^2\theta} \\
F_\parallel &= F_\perp \frac{a^2 - 2a\sin\theta\tan\theta + \sin^2\theta\tan^2\theta}{a^2 + 2a\sin\theta\tan\theta + \sin^2\theta\tan^2\theta} \\
a &= \sqrt{\eta^2 - \sin^2\theta}
\end{aligned}$$

The total reflectance F for polarized light is a weighted sum of the two components, $a_0F_\perp + a_1F_\parallel$, where the weights a_0 and a_1 sum to unity. For unpolarized light, $F = (F_\perp + F_\parallel)/2$.

The `D3DXFresnelTerm` function computes the Fresnel reflection term F for unpolarized light, incident at an angle whose cosine is given by the `cos_angle` parameter, reflecting from an interface between two simple materials whose relative index of refraction is given by the `refraction_index` parameter.

```
float ::D3DXFresnelTerm(float cos_angle, float refraction_index);
```

15.17 Error Handling

DirectX provides some error handling facililites that are global across all DirectX components. The functions `DXGetErrorDescription8` and `DXGetErrorString8` are used to retrieve strings that correspond to failed `HRESULT`s. `DXGetErrorString8` returns a simple string corresponding to the failed HRESULT, such as "D3DERR_DEVICELOST". DXGetErrorDescription8 returns a string that describes the error, such as "device lost". The global error functions require the inclusion of <dxerr9.h> and the linking of dxerr9.lib.

```
LPCTSTR ::DXGetErrorString9(HRESULT hr);
LPCTSTR ::DXGetErrorDescription9(HRESULT hr);
```

In addition to these global error functions, D3DX provides a similar function, D3DXGetErrorString, for Direct3D and D3DX errors. Instead of returning the string as the result of the function, the caller supplies a buffer into which the string is written.

```
HRESULT ::D3DXGetErrorString(HRESULT hr,
            LPTSTR pBuffer,
            UINT BufferLen);
```

The following HRESULT error codes are specific to D3DX.

```
enum _D3DXERR
{
    D3DXERR_CANNOTATTRSORT          = MAKE_DDHRESULT(2902),
    D3DXERR_CANNOTMODIFYINDEXBUFFER = MAKE_DDHRESULT(2900),
    D3DXERR_INVALIDDATA             = MAKE_DDHRESULT(2905),
    D3DXERR_INVALIDMESH             = MAKE_DDHRESULT(2901),
    D3DXERR_LOADEDMESHASNODATA      = MAKE_DDHRESULT(2906),
    D3DXERR_SKINNINGNOTSUPPORTED    = MAKE_DDHRESULT(2903),
    D3DXERR_TOOMANYINFLUENCES       = MAKE_DDHRESULT(2904)
};
```

15.18 ID3DXBuffer

Several functions and object methods in D3DX, the mesh functions and methods in particular, need a way to pass an arbitrarily sized and typed buffer between the application and the function or object. The ID3DXBuffer encapsulates this concept as a COM object that implements a simple wrapper around a variable-sized chunk of memory. An application can create a buffer object with the D3DXCreateBuffer function. The object allocates size bytes and manages the lifetime of the allocated memory with the lifetime of the object.

```
HRESULT ::D3DXCreateBuffer(DWORD size,
            ID3DXBuffer **result);
```

The ID3DXBuffer interface includes only two methods: one for obtaining a pointer to the allocated memory and one for obtaining the size of the allocated memory.

Interface 15.1: Summary of the `ID3DXBuffer` interface.

ID3DXBuffer

Read-Only Properties	
`GetBufferPointer`	A pointer to the contained data
`GetBufferSize`	The size of the contained data

```
interface ID3DXBuffer : IUnknown
{
    void *GetBufferPointer();
    DWORD GetBufferSize();
};
```

The type of the data contained in the buffer is not exposed by the interface. When we know the type of the encapsulated data, we can use a template-based wrapper to safely expose the underlying type of the contained data. The class `rt::dx_buffer` in the include file `rt/mesh.h` in the sample code contains such a wrapper.

15.19 Vertex Declarations

D3DX provides several functions for manipulating vertex FVF codes and their corresponding vertex shader declarations. `D3DXGetFVFVertexSize` returns the size in `BYTE`s of a vertex, calculated from its flexible vertex format code. `D3DXDeclaratorFromFVF` returns the vertex shader declaration corresponding to an FVF code. Every FVF code can be expressed as a vertex shader declaration. `D3DXFVFFromDeclarator` returns an FVF code corresponding to the given vertex shader declaration, if any. FVF codes are a subset of vertex shader declarations, so this function may fail. The shader declaration token array should contain an end token, or be at least as big as `MAX_FVF_DECL_SIZE`

```
UINT ::D3DXGetFVFVertexSize(DWORD fvf);

HRESULT ::D3DXDeclaratorFromFVF(DWORD fvf,
    DWORD result[MAX_FVF_DECL_SIZE]);

HRESULT ::D3DXFVFFromDeclarator(const DWORD *declaration,
    DWORD *result);
```

15.20 Intersection Testing

D3DX provides functions for computing the intersection of a ray with a triangle with `D3DXIntersectTri`. The coordinates of the triangle vertices are given by the p0, p1, and p2 parameters. The starting point of the ray is given by the position parameter and the ray's direction is given by the direction parameter. The result of the function is TRUE when the ray intersects the triangle and FALSE otherwise. If the ray intersects the triangle, then the barycentric coordinates of the intersection point are returned in the u and v parameters. The distance of the intersection point from the origin of the ray is returned in the distance parameter.

```
BOOL ::D3DXIntersectTri(const D3DXVECTOR3 *p0,
    const D3DXVECTOR3 *p1,
    const D3DXVECTOR3 *p2,
    const D3DXVECTOR3 *position,
    const D3DXVECTOR3 *direction,
    float *u,
    float *v,
    float *distance);
```

D3DX also provides the D3DXSphereBoundProbe and D3DXBoxBoundProbe functions to test the intersection of a ray with a bounding volume. This is most useful in picking objects on the screen. First, the pick ray can be intersected with an object's bounding volume. If the pick ray doesn't intersect the object's bounding volume, then it is not necessary to test the triangles in the object for intersection, avoiding many unnecessary triangle itnersection tests.

A bounding sphere is given by the coordinate of the center of the sphere and its radius. A bounding box is given by two coordinates representing the minimum and maximum extent of the box. In each case, the ray is given as a position and direction and the result of the function is TRUE if the ray intersects the volume, as in D3DXIntersectTri.

```
BOOL ::D3DXSphereBoundProbe(const D3DXVECTOR3 *center,
    float radius,
    const D3DXVECTOR3 *position,
    const D3DXVECTOR3 *direction);

BOOL ::D3DXBoxBoundProbe(const D3DXVECTOR3 *minima,
    const D3DXVECTOR3 *maxima,
    const D3DXVECTOR3 *position,
    const D3DXVECTOR3 *direction);
```

15.21 Shader Assembly

D3DX provides functions for converting shader instructions in text form into an array of shader tokens. The assembly process can also perform a validation step on the shader to ensure that the shader instructions do not violate any of the appropriate rules for constructing vertex and pixel shaders.

The functions `D3DXAssembleShader`, `D3DXAssembleShaderFromFile` and `D3DX-AssembleShaderFromResource` assemble a pixel or vertex shader from ASCII text located in a block of memory, a file on disk, or a Win32 executable module resource, respectively. If the shader is encoded with Unicode characters, it should be converted to ANSI before using these functions. In each function, three `ID3DXBuffer` objects are returned to the caller containing vertex shader constant declarations, the assembled shader token array, and an array of strings containing any assembly warning or error messages. Any of these arguments may be `NULL` if the associated data is not needed by the caller.

```
HRESULT ::D3DXAssembleShader(const void *data,
            UINT size,
            DWORD flags,
            ID3DXBuffer **constants,
            ID3DXBuffer **shader,
            ID3DXBuffer **errors);

HRESULT ::D3DXAssembleShaderFromFile(LPCTSTR filename,
            DWORD flags,
            ID3DXBuffer **constants,
            ID3DXBuffer **shader,
            ID3DXBuffer **errors);

HRESULT ::D3DXAssembleShaderFromResource(HMODULE module,
            LPCTSTR resource,
            DWORD flags,
            ID3DXBuffer **constants,
            ID3DXBuffer **shader,
            ID3DXBuffer **errors);
```

The `flags` argument can be zero or more of the following flags to indicate options to the assembly process. `D3DXASM_DEBUG` inserts debugging information as comments into the assembled shader. `D3DXASM_SKIPVALIDATION` does not perform any validation checks on the assembled shader.

```
#define D3DXASM_DEBUG          (1 << 0)
#define D3DXASM_SKIPVALIDATION (1 << 1)
```

15.22 Further Reading

Principles of Digital Image Synthesis by Andrew Glassner gives more information about the interaction between light and matter and how this relates to computer graphics. This two-volume work is a definitive reference for the physics of advanced lighting and shading and materials.