Part I

Preliminaries

Chapter 1

Introduction

1.1 Overview

This book describes the Direct3D graphics pipeline, from presentation of scene data to pixels appearing on the screen. The book is organized sequentially following the data flow through the pipeline from the application to the light emitted from the CRT monitor. Each major section of the pipeline is treated by a part of the book, with chapters and subsections detailing each discrete stage of the pipeline. This section summarizes the contents of the book.

Part I begins with a review of basic concepts used in 3D computer graphics and their representations in Direct3D. The `IDirect3D8` interface is introduced and device selection is described. The `IDirect3DDevice8` interface is introduced and an overview of device methods and internal state is given. Finally, a basic framework is given for a 2D application.

Chapter 1 begins with an overview of the entire book. A review is given of display technology and the important concept of gamma correction. The representation of color in Direct3D and the macros for manipulating color values are described. The relevant mathematics of vectors, geometry and matrices are reviewed and summarized. A summary of **COM** and the `IUnknown` interface is given. Finally, the coding style conventions followed in this book are presented along with some useful C++ coding techniques.

Chapter 2 describes the Direct3D object. Every application instantiates this object to select a device from those available. Available devices advertise their location in the virtual desktop and their capabilities to applications through the Direct3D object. Selecting a device from those available and examining a device's capabilities are described. Multiple monitor considerations are also discussed.

Chapter 3 describes the Direct3D device object which provides access to the rendering pipeline. The device is the interface an application will use most often and it has a large amount of internal state that controls every stage of the rendering pipeline. This chapter provides a high-level overview of the device

and its associated internal state. Detailed discussion of the device state appears throughout the rest of the book.

Chapter 4 describes the basic architecture of a typical Direct3D application. Every 3D application can use 2D operations for manipulating frame buffer contents directly. An application can run in full-screen or windowed modes and the differences are presented here. The handling of Windows messages and a basic display processing loop are presented. At times it may be convenient to use GDI in a Direct3D application window and a method for mixing these two Windows subsystems is presented. Almost every full-screen application will want to use the cursor management provided by the device. Device color palettes and methods for gamma correction are presented.

Part II describes the geomtry processing portion of the graphics pipeline. The application delivers scene data to the pipeline in the form of geometric primitives. The pipeline processes the geometric primitives through a series of stages that results in pixels displayed on the monitor. This part describes the start of the pipeline where the processing of geometry takes place.

Chapter 5 describes how to construct a scene representing the digital world that is imaged by the imaginary camera of the device. A scene consists of a collection of models drawn in sequence. Models are composed of a collection of graphic primitives. Graphic primitives are composed from streams of vertex and index data defining the shape and appearance of objects in the scene. Vertices and indices are stored in resources created through the device.

Chapter 6 covers vertex transformations, vertex blending and user-defined clipping planes. With transformations, primitives can be positioned relative to each other in space. Vertex blending, also called "skinning", allows for smooth mesh interpolation. User-defined clipping planes can be used to provide cutaway views of primitives.

Chapter 7 covers viewing with a virtual camera and projection onto the viewing plane which is displayed as pixels on the monitor. After modeling, objects are positioned relative to a camera. Objects are then projected from 3D camera space into the viewing plane for conversion into 2D screen pixels.

Chapter 9 covers programmable vertex shading. Programmable vertex shaders can process the vertex data streams with custom code, producing a single vertex that is used for rasterization. The vertex shading machine architecture and instruction set are presented.

Part III covers the rasterization portion of the pipeline where geometry is converted to a series of pixels for display on the monitor. Geometric primitives are lit based on the lighting of their environment and their material properties. After light has been applied to a primitive, it is scan converted into pixels for processing into the frame buffer. Textures can be used to provide detailed surface appearance without extensive geometric modeling. Pixel shaders can be used to provide custom per-pixel appearance processing instead of the fixed-function pixel processing provided by the stock pipeline. Finally, the pixels generated from the scan conversion process are incorporated into the render target surface by the frame buffer.

Chapter 8 describes the lighting of geometric primitives. The lighting model

is introduced and the supported shading algorithms and light types are described.

Chapter 10 describes the scanline conversion of primitives into pixel fragments. Lighting and shading are used to process vertex positions and their associated data into a series of pixel fragments to be processed by the frame buffer.

Chapter 11 describes textures and volumes. Textures provide many efficient per-pixel effects and can be used in a variety of manners. Volumes extend texture images to three dimensions and can be used for a volumetric per-pixel rendering effects.

Chapter 12 describes programmable pixel shaders. Programmable pixel shaders combine texture map information and interpolated vertex information to produce a source pixel fragment. The pixel shading machine architecture and instruction set are presented.

Chapter 13 describes how fragments are processed into the frame buffer. After pixel shading, fragments are processed by the fog, alpha test, depth test, stencil test, alpha blending, dither, and color channel mask stages of the pipeline before being incorporated into the render target. A render target is presented for display on the monitor and video scanout.

Part IV covers the D3DX utility library. D3DX provides an implementation of common operations used by Direct3D client programs. The code in D3DX consists entirely of client code and no system components. An application is free to reimplement the operations provided by D3DX, if necessary.

Chapter 14 introduces D3DX and summarizes features not described elsewhere.

Chapter 15 describes the abstract data types provided by D3DX. D3DX provides support for RGBA color, point, vector, plane, quaternion, and matrix data types.

Chapter 16 describes the helper COM objects provided by D3DX. D3DX provides a matrix stack object to assist in rendering frame hierarchies, a font object to assist in the rendering of text, a sprite object to assist in the rendering of 2D images, an object to assist in rendering to a surface or an environment map and objects for the rendering of special effects.

Chapter 17 describes the mesh objects provided by D3DX. The mesh objects provided by D3DX encompass rendering of indexed triangle lists as well as progressive meshes, mesh simplification and skinned meshes.

Part V covers application level considerations. This part of the book describes issues that are important to applications but aren't strictly part of the graphics pipeline. Debugging strategies for applications are presented. Almost all Direct3D applications will be concerned with performance; API related performance issues are discussed here. Finally, installation and deployment issues for Direct3D applications are discussed.

Chapter 19 describes debugging strategies for Direct3D applications. This includes using the debug run-time for DirectX 8, techniques for debugging full-screen applications and remote debugging.

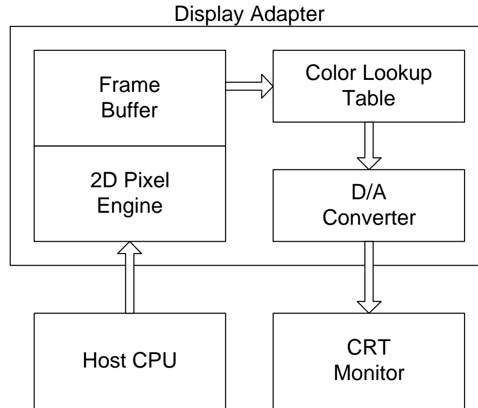

Figure 1.1: Subsystems of a typical 2D graphics adapter.

Chapter 20 covers application performance considerations. All real devices have limitations that affect performance. A general consideration of how the pipeline state affects performance is given.

Chapter 21 covers application installation and setup.

Appendix A provides a guided tour of the DirectX SDK materials.

1.2 Display Technology

The typical Windows computer display system consists of a single graphics adapter and a display monitor. The graphics adapter contains the hardware needed store and display images on a CRT monitor or other display device. The graphics adapter in PC compatibles has evolved from a monochrome character display to a color character display, to simple memory-mapped frame buffers, to devices with 2D accelerators to more recent devices capable of sophisticated 3D realism. All of these adapters stored a representation of the displayed image in a bank of dual-ported memory called a **frame buffer**. The system uses one of the ports to read and write images into the frame buffer. The second port is used by the video scanout circuitry of the adapter to create a signal for the monitor. See figure 1.1 for a schematic representation. PC adapters can typically operate in either character or graphics mode.

Monochrome monitors can only display white or black. Grayscale monitors can display varying shades of gray from black to white. Color monitors can display a variety of hues and shades. Synthetic imagery typically uses a color monitor, but can also be used with a grayscale monitor. A monochrome

monitor usually requires a monochrome adapter. Direct3D does not support monochrome or grayscale adapters, only color VGA compatible adapters.

When used in graphics mode, the adapter's memory represents a 2D grid of independently addressable **pixels**. The number of bits in the pattern used to define each pixel is referred to as the screen depth. Three dimensional graphics generally requires a pixel depth of 16 bits or more to achieve the desired color fidelity. The bit pattern stored in memory can encode the pixel's color information directly, or it can encode an index into a **CLUT**. These are the so-called direct and indexed color display modes. Direct3D8 supports direct color modes whose bit depth is 16, 24, or 32 bits.

In direct color mode, the bit pattern is scanned out of memory and directly used to supply inputs to a **DAC** to produce the analog video signal sent to the monitor. The bit pattern may be passed through a gamma correction CLUT before being sent to the DAC.

Most desktop monitors are **CRT** monitors. CRT monitors produce an image with a modulated electron beam that scans across the screen to excite phosphor dots on the interior surface of the picture tube. The electron beam is produced by an an **electron gun** and scans the screen in a regular grid.

The phosphors on a CRT are grouped in clusters of three types: red, green and blue phosphors. When the electron beam scans the phosphor dots, the phosphor emits light. The chemical composition of the phosphor dots is chosen to approximate the ideal red, green and blue color primaries that are added to produce the desired color.

The electron beam can be scanned across the screen in either an **interlaced** or **non-interlaced** fashion. Television monitors use an interlaced, or progressive, scanning pattern. VGA monitors use a non-interlaced scanning pattern. An interlaced scanning pattern consists of two fields, one field displaying the odd numbered scanlines and the other field displaying the even numbered scanlines.

A non-interlaced pattern scans the screen starting at the top left of the screen and scanning to the right, exciting phosphors one **scanline** at a time. The phosphor triplets are excited with an electron beam whose intensity is modulated by the voltage of the analog video signal driving the monitor.

When the right edge of the screen is reached, the electron beam is turned off during **horizontal retrace** to start the next scanline at the left edge of the screen. When the bottom of the screen is reached, the electron beam is turned off during **vertical retrace** to reposition the beam at the top left corner of the screen.

Rather than scan all three phosphor dots with a single electron gun, a CRT will usually include three electron guns, one for each primary, scanning in parallel. A **shadow mask** placed between the picture tube surface and the electron guns blocks the electron beams from illuminating phosphors other than those corresponding to the currently addressed pixel.

So far we have described an ideal CRT that is perfectly addressable. An ideal CRT would illuminate a single phosphor triplet perfectly with no other phosphors receiving excitation from the electron beam. A real-world monitor does not address an individual phosphor triplet directly, but excites a shadow-

masked swath as it traverses the screen. The phosphors are not perfect primary color sources. The glass of the tube itself can scatter a phosphor's emitted light. All of these factors and others combine to distort the desired image scanned out from the frame buffer. The next section discusses a way to compensate for this distortion.

1.3 Gamma Correction

The intensity of light resulting from the phosphors excited by an electron gun is not linear with respect to the voltage applied to the electron gun. If two voltages are applied with the second voltage being twice the first, the resulting second intensity will not necessarily be twice the first intensity. Synthetic imagery assumes that colors can be manipulated linearly as additive primaries, requiring compensation for the non-linearity of the monitor.

The non-linearity of a display is caused by many factors, such as the point spread function, the shadow mask, the phosphor's composition and even the age of the components in the monitor itself. The non-linearity can be approximated by the equation $I = kV^\gamma$ relating the intensity I to the applied voltage V. The term **gamma correction** derives from the use of the greek letter gamma in this equation.

A display may be characterized by a single value for γ. A more accurate characterization accounts for the differences in the emission properties of the phosphors and treats each primary separately, modeling each primary's response with its own power law and γ value.

To ensure the best results from synthetic imagery, a monitor should be calibrated and its gamma measured by experiment. Poor compensation of gamma can have a deleterious effect on antialiasing algorithms and cause imagery to appear "too dark" or "too washed out".

Gamma correction is most easily implemented as a color lookup table applied just before the data is presented to the DAC. Some display adapters do not provide such a DAC and it must be incorporated into the application's processing, at the expense of some color accuracy.[1]

Gamma Correction with GDI

Windows GDI provides two functions for getting and setting the gamma ramp on a GDI device. This can be used when a Direct3D application is running in windowed mode, however this method is not well supported by GDI device drivers. In Windows 2000, you can query a GDI device context for support of these functions by examining the `CM_GAMMA_RAMP` bit of the `COLORMGMTCAPS`.

For more information see the Platform SDK help on the functions `::GetDeviceGammaRamp`, `::SetDeviceGammaRamp` and `::GetDeviceCaps`.

[1] Lighting and shading all assume a linear colorspace, applying gamma correction at the application layer presents non-linear color information to the pipeline, which it manipulates as linearly related information.

Measuring Gamma Interactively

You can interactively measure the gamma of your monitor. When viewed from a distance, a rectangular region filled with alternating scanlines of black (minimum intensity) and white (maximum intensity) will appear to be a uniform area at 50% intensity. The perceived intensity will appear equal to a properly gamma-corrected solid rectangle drawn at 50% intensity.

To determine the gamma interactively, have the user adjust a slider control adjusting the gamma until the two rectangle drawn with these techniques appear of equal intensity when viewed at a distance. The gamma value of the display is the gamma value corresponding to the slider's position when a match is found. To measure the gamma of the display primaries separately, perform a similar task, but use alternating scanlines of black and full intensity of the primary.

The `rt_Gamma` sample on the accompanying CD-ROM uses this algorithm to measure a monitor's gamma directly with Direct3D.

Gamma Correction with Direct3D

First, an application should decide how they want to deal with device nonlinearity. Monitor gammas vary enough that you will want to provide some sort of compensation. The best way to deal with the problem is to correct for the monitor's gamma during video scanout with the device's gamma ramp in exclusive mode. All color inputs supplied to the Direct3D device, including texture and volume inputs, are provided in a linear color space with a γ of 1.0.

In exclusive mode, the gamma ramp of the device can be used to correct for the monitor's nonlinearity. In windowed mode, an application can use the GDI support. Because of the global affect of the GDI gamma ramp on all applications on the same monitor, you may prefer to deal with gamma in a less intrusive manner. Lack of hardware gamma ramps or driver support could also force you to deal with gamma in another manner.

An application can take its linear color inputs and manually gamma correct them, including textures and volumes, when they are loaded into the device. This still leads to an inaccurate rendering since the deive is performing operations on colors it assumes to be related linearly, but it is better than no compensation for monitor gamma.

1.4 Color

We usually have an intuitive idea of what we mean by color. In computer graphics we must be quantitative and precise when we describe color to the computer. In addition, there are factors due to the human visual system that must be considered when choosing color, as artists have long well known. The human visual system can be considered the very last stage of the graphics pipeline after the monitor produces an image.

A color is described to the computer as an ordered tuple of values from a **color space**. The values themselves are called **components** and are coordi-

nates in the color space. Windows GDI represents colors as an ordered tuple of red, green and blue components with each component in the range $[0.0, 1.0]$ represented as an unsigned byte quantity in the range $[0, 255]$.

By default Windows GDI uses the sRGB color space[2]. This is an adopted international standard color space that was first proposed by Hewlett-Packard and Microsoft. However, the sRGB space is device-dependent owing to the non-linearities of display systems.

Other color spaces provide device-independent color specifications. In 1931, the **CIE** created a device-independent standard color space CIEXYZ. There are variations of the CIE color space such as CIELUV, CIELAB, and others. This colorspace can be used to ensure accurate reproduction of colors across a variety of output technologies, such as matching screen output to printed output.

In computer graphics, it is often convenient to use the **HLS** and **HSV** color spaces for creating color ramps. If the intent of your application is to map data to a range of perceptibly equal colors, you will want to use a color space other than sRGB. Equal increments in the components of an sRGB color do not give rise to equal increments in perceived color change on the monitor.

A full treatment of color and its perception is beyond the scope of this book. The mechanics of color matching and color space manipulation on Windows can be found in the Platform SDK Help on Image Color Management (ICM). The perceptual and artistic aspects of color matching and color spaces can be found in [Thorell] and [Hall].

Transparency With Alpha

Many times in computer graphics we wish to combine pixels as though they were painted on layers of mylar and a light was shown through them as in traditional cel animation. In Direct3D transparency is represented as an additional channel of information representing the amount of transparency of the pixel.

When a pixel is fully opaque, its alpha value is 1.0 and this pixel completely obscures anything behind it. When a pixel is fully transparent, its alpha value is 0.0 and everything behind the pixel shows through. When the alpha value is between 0 and 1, then the pixel is partially transparent. A pixel's alpha value can be used to combine the foreground of the pixel with some background based on the following formula:

$$C' = \alpha C_f + (1 - \alpha) C_b$$

When $\alpha = 0$, the resulting color C' contains no amount of the foreground color C_f and all of the background color C_b. When $\alpha = 0.5$, the resulting color contains an equal mix of C_f and C_b. When $\alpha = 1.0$, the resulting color contains only the foreground color C_f and no amount of the background color C_b.

A pixel's alpha channel can be also used for generalized masking and matte effects in addition to simple transparency. In these cases a different formula

[2] http://www.color.org/sRGB.html

is used to combine foreground and background pixels. The alpha of a pixel is independent of the colorspace from which the pixel is drawn.

Color in Direct3D

Direct3D uses the sRGB color space in windowed mode when GDI owns the screen. In full-screen mode, an application can use a linear RGB color space if gamma correction is available, otherwise the application will have to provide gamma corrected source colors to Direct3D to achieve linear color response. The linear RGB color space comes at the expense of some dynamic range when color channels are 8 bits wide, but provides for the most accurate rendition of the desired colors on the display.

Direct3D uses a variety of representations for colors depending on the context in which they are used. When Direct3D accepts a parameter to the rendering pipeline, those parameters are specified either as a single `DWORD`, or as a struct of 4 floats. When interacting with image surface memory on a device, the colors will be in a device-specific representation.

A struct of floating-point values is used in lighting and shading calculations where accuracy is important. `DWORD`s are generally used in pixel related operations. Floating-point component values typically fall into the range $[0.0, 1.0]$.[3]

When working with GDI DIBs, 24-bit RGB colors packed in a `DWORD` sized quantity are common. GDI provides `COLORREF` and `PALETTEENTRY` types for representing colors. Direct3D also stores its colors in a `DWORD` sized quantity, but whereas GDI uses 8 bits of the `DWORD` for flags in a `PALETTEENTRY` and is ignored in an RGB `COLORREF`, Direct3D uses those 8 bits for an alpha channel.

The typedef `D3DCOLOR` introduces an alias for a `DWORD` and the macros[4] in table 1.1 are provided for manipulating color in this representation. The `D3DCOLOR_XRGB` macro provides an alpha value of 1.0.

The `D3DCOLOR_VALUE` structure contains an RGBA color as four floating-point values:

```
typedef struct _D3DCOLORVALUE
{
    float r;
    float g;
    float b;
    float a;
} D3DCOLORVALUE;
```

The D3DX utility library defines a `D3DXCOLOR` structure with member functions and support functions for color manipulation. It is described in chapter 14.

[3] Certain special effects exploit the use of color in the pipeline with values outside this range.

[4] The macros are presented as if they were inline functions with prototypes, but no type checking is performed with macros.

Type Definition
`typedef DWORD D3DCOLOR;`

Macros
`D3DCOLOR D3DCOLOR_ARGB(BYTE a, BYTE r, BYTE g, BYTE b)`
`D3DCOLOR D3DCOLOR_RGBA(BYTE r, BYTE g, BYTE b, BYTE a)`
`D3DCOLOR D3DCOLOR_XRGB(BYTE r, BYTE g, BYTE b)`
`D3DCOLOR D3DCOLOR_COLORVALUE(float r, float g, float b, float a)`

Table 1.1: 32-bit RGBA `D3DCOLOR` macros.

1.5 Basic Geometry

Points

In geometry, a point is represented by its coordinate in space. Geometry usually uses a Cartesian coordinate system, where the coordinates of a point in space are represented by the distance along each of the principal axes to the point. While other coordinate systems are used in geometry (such as spherical or cylindrical coordinate systems), 3D graphics uses a Cartesian coordinate system. The dimensionality of a point corresponds to the number of coordinates needed to represent the point. Thus a two dimensional point requires two Cartesian coordinates, and a three dimensional point requires three coordinates.

Lines, Rays and Segments

A line has direction and is infinite in length. The direction line be defined by two points through which the line passes. Two or more points all on a line are said to be colinear. Because a line is defined by two points, two points are always colinear.

A ray begins at a point and extends infinitely in a direction away from the point. A ray is defined by a point and a direction.

A line segment is a finite length line defined by its two endpoints.

In computers we must deal with finite quantities, so we can't draw the full extent of a line or ray according to its mathematical definition. In computer graphics when referring to "lines" we most often mean line segments.

Planes and Triangles

A plane is an oriented sheet in 3-space with no thickness and an infinite extent. A plane is defined by three non-colinear points that intersect the plane or by a point on the plane and a direction perpendicular to the plane. The direction perpendicular to a plane is called the normal to the plane.

A triangle is also defined by three points in 3-space called vertices (singular vertex). The triangle is to the plane what the line segment is to the line. A

triangle encloses a finite area defined by interior region bounded by the line segments that join its vertices.

1.6 Homogeneous Coordinates

Homogeneous coordinates are often used in computer graphics. Homogeneous coordinates can be thought of as extending Cartesian coordinates to include the concept of infinity. The extension is performed by adding an additional coordinate to the Cartesian coordinates. This additional coordinate is often represented as w. To convert a point from its homogeneous representation to its Cartesian representation, the coordinates are divided by w, thus (x, y, z, w) becomes $(x/w, y/w, z/w, 1)$. The final coordinate can be discarded to retrieve the corresponding Cartesian coordinate. You can see that if $w = 0$, then the division yields a Cartesian point with infinite coordinates.

In homogeneous coordinates, all points with $w = 0$ are at infinity. If we imagine the 2D case of points on a plane we can see that "infinity" lies in many different directions since a plane has infinite extent in all directions. Homogeneous coordinates allow us to differentiate between the different "directions of infinity". Thus the 2D homogeneous point $(1, 0, 0)$ is infinity in the direction of the positive x axis, while $(0, 1, 0)$ represents infinity in the direction of the positive y axis.

Homogeneous coordinates can also be used to define projective mappings. The conversion of a homogeneous coordinate into its Cartesian equivalent is a projective mapping from $N+1$ dimensions to N dimensions for N dimensional Cartesian space. Projective mappings occur in 3D computer graphics when we project a volume of imaginary space onto the screen of the monitor. Projective mappings also occur when using texture.

1.7 Vectors

A **vector** has length and an orientation in space. Unlike a line, a vector does not have a position in space. Two vectors are equal if they have equal length and the same orientation. Vectors of length 1.0 are called **unit vectors**. A **normal** vector is a unit vector that is perpendicular to an object's surface, pointing towards the "outside" of the surface. Vectors will be written with a lower case letter and an arrow accent: $\vec{i}$, $\vec{j}$, and $\vec{k}$ will denote unit vectors along the three principal axes in a 3D coordinate system. A vector's scalar components are shown as $\langle x,\ y,\ z \rangle$ or $\langle v_0,\ v_1,\ v_2 \rangle$. Vector arithmetic is summarized in table 1.2.

Direct3D represents 3D vectors using the `D3DVECTOR` structure. The D3DX utility library also includes the 2D, 3D and 4D vector classes `D3DXVECTOR2`, `D3DXVECTOR3` and `D3DXVECTOR4`, respectively. The 3D vector class inherits from `D3DVECTOR`. The D3DX vector class provide member functions for vector construction, element access, and arithmetic operators. They are described in chapter 14.

Addition	$\vec{a} + \vec{b} = \langle a_x + b_x,\ a_y + b_y,\ a_z + b_z \rangle$
Scalar Multiplication	$s\vec{a} = \langle sa_x,\ sa_y,\ sa_z \rangle$
Length	$\lVert \vec{a} \rVert = \sqrt{a_x^2 + a_y^2 + a_z^2}$
Scalar Dot Product	$\vec{a} \cdot \vec{b} = \langle a_x b_x,\ a_y b_y,\ a_z b_z \rangle$
Vector Cross Product	$\vec{a} \otimes \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = \langle a_y b_z - b_y a_z,\ b_x a_z - a_x b_z,\ a_x b_y - b_x a_y \rangle$

Table 1.2: Vector operations given $\vec{a} = \langle a_x,\ a_y,\ a_z \rangle$ and $\vec{b} = \langle b_x,\ b_y,\ b_z \rangle$.

1.8 Coordinate Systems

Scenes are described geometrically to Direct3D in a coordinate system. Here we will summarize coordinate systems in an informal manner. For many 3D graphics problems, it is helpful to have a working knowledge of 2D and 3D geometry. The *Graphics Gems* series of books contain many useful geometric and algorithm results for manipulating geometry. The book series has been discontinued and replaced by the *Journal of Graphics Tools*, following the same style used by the books.

Positions are defined by **points** drawn from a **space**. A point is always associated with a particular space, or **coordinate frame**. The scalar components of a point are the values of the coordinate axes in the space from which it was drawn. Points will be denoted $\mathrm{P}(x, y, z)$ with capital letters, optionally followed by their coordinate values in parenthesis.

In computer graphics, two dimensional space usually refers to a screen coordinate system. **Screen coordinates** place the origin in the upper left corner of the screen, with increasing x to the right and increasing y to the bottom. Screen coordinates are the default coordinate system in GDI and are also used in some screen-oriented aspects of Direct3D.

Three dimensional space is defined by three mutually perpendicular axes. Two common orientations of 3D axes are **left-handed** and **right-handed** coordinate systems. The name stems from a simple process used to visualize the coordinate systems. First, point the fingers of the right hand in the direction of the positive x axis. Curl the fingers inward towards the palm by rotating them towards the positive y axis. The thumb will then point in the positive z direction in a right-handed coordinate system. For a left-handed coordinate system the thumb will be pointing in the negative z direction. The results are reversed if you use your left hand, see figure 1.2.

In computer graphics, a right-handed coordinate system with the origin in the lower left corner of the screen and positive x increasing to the right and positive y increasing to the top, the z axis points out of the screen towards the viewer. In a left-handed coordinate system with a similar arrangement of the x and y axes, the z axis points into the screen away from the viewer. Direct3D uses

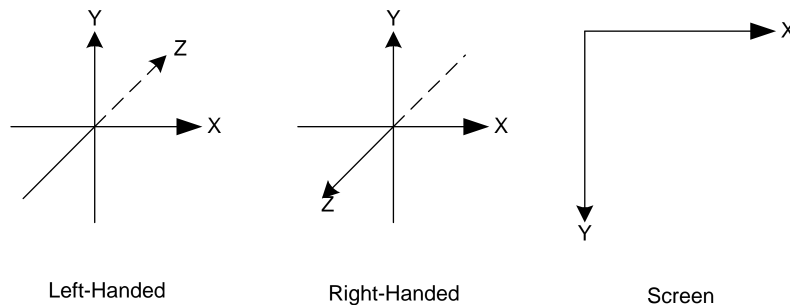

Figure 1.2: Left-handed, right-handed and screen coordinate systems. The xy plane is in the plane of the page and the dashed portion of the z axis extends behind the page, while the solid portion of the z extends in front of the page.

a left-handed coordinate system, while OpenGL uses a right-handed coordinate system. When converting algorithms from other graphics APIs to Direct3D, you must take the "handedness" of the API into consideration.

Three dimensional points in Direct3D are described with the `D3DVECTOR` structure.[5] Two dimensional points can be represented as 3D points with a fixed value for the z component.

Geometric **lines** are infinite in length and have an orientation and a position. Computer graphics generally deals with **line segments** which are of finite length, but generally refers to them as "lines". A **ray** has one finite end point and extends infinitely in the direction of the ray. Direct3D provides no data structure for storing a line, line segment, or ray, but these are easily created if needed in your application.

When two lines intersect, they define a **plane** in space. The D3DX utility library represents planes with the `D3DXPLANE` structure. A set of member functions and global functions are provided for manipulating planes. It is described in chapter 14.

1.9 Matrices

Matrices map one coordinate system to another coordinate system and are said to transform coordinates from one coordinate system to another. Transformation matrices in computer graphics are typically small, rectangular arrays of floating-point values, with 2x2, 3x3 and 4x4 being the most commonly used sizes of matrices. Direct3D declares a matrix as a 4x4 array of floats. The first subscript increases from top to bottom along the matrix elements and the second subscript increases from left to right along the matrix elements:

[5] Points are not provided with a distinct type.

Identity	$\mathbf{I} = \mathbf{M}$, where $m_{ij} = 1$ if $i = j$, else 0
Inverse	$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$
	$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$
Transpose	$\mathbf{A}^T = \mathbf{M}$, where $m_{ij} = a_{ji}$
Addition	$\mathbf{A} + \mathbf{B} = \mathbf{M}$, where $m_{ij} = a_{ij} + b_{ij}$
Scalar Product	$s\mathbf{A} = \mathbf{M}$, where $m_{ij} = sa_{ij}$
Vector Product	$\vec{v}\mathbf{A} = \langle m_0, m_1, m_2, m_3 \rangle$, where $m_i = \vec{v} \cdot A_i$
Matrix Product	$\mathbf{AB} = \mathbf{M}$, where $m_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

Table 1.3: Matrix operations given two n by n matrices **A** and **B**.

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$$

Matrix arithmetic operations are summarized in table 1.3. Note that matrix multiplication is not commutative, so that $\mathbf{AB} \neq \mathbf{BA}$ for arbitrary **A** and **B**. The former is referred to as **post-multiplying B** onto **A** and the latter is referred to as **pre-multiplying B** onto **A**.

Sometimes the inverse of a matrix is needed in computer graphics. While the procedure for obtaining the inverse of an arbitrary matrix is complex, the matrices used in computer graphics are often **affine transformations**. Affine transformations have the property that their inverse is their transpose, which is trivial to compute.

Direct3D provides the `D3DMATRIX` structure for storing transformation matrices. D3DX provides a `D3DXMATRIX` class that inherits from `D3DMATRIX` providing basic mathematical operations as C++ operators and a set of functions for operating on matrices. See chapter 14.

1.10 Aliasing

Aliasing arises as a consequence of the discrete sampling of continuous signals. In computer graphics, discrete samples are everwhere – scenes are rendered by taking discrete samples of geometric objects at each pixel on the screen, texture images are 2D collections of discrete samples of an object's surface appearance, texture volumes are 3D collections of discrete samples distributed over space, applying 2D and 3D textures during rendering samples the underlying texture, colors are sampled into a color space, lighting is sampled at the vertices of a polygon and so on.

Aliasing is most easily observed by most algorithms used to draw lines whose slope is not a multiple of 45 degrees. The effect is most observable on a display when the line is slightly misaligned with respect to the horizontal or vertical axis of the pixel (sample) grid. The "stair-step" effect seen in the pixel pattern

is the result of aliasing introduced by the discrete screen pixel locations. The points on a geometric line have a continuously varying position along the line and no thickness. The points on a rasterized line can only be located at the pixel locations. Further, a pixel is not dimensionless like a geometric point; a pixel has area. The restrictions of the pixel grid and rasterized lines and other graphics primitives, when compared to their geometric counterparts, are what give rise to aliasing. This is most prominent on object edges where the silhouette of the object shows the staircase pattern and in scenes containing many finely spaced line patterns which give rise to Moiré patterns.

As well as spatial aliasing, dynamic rendering of scenes can result in **temporal aliasing**. Motion pictures can exhibit temporal aliasing as an interaction between fast-moving objects in the scene and the shutter (sampling) mechanism of the camera. This is most commonly seen in the wagon wheel spokes of westerns; the strobing action of the camera shutter can make the apparent speed of the wheel slow down, stop, or even reverse. Similar effects occur when animation is used.

Aliasing can appear in any value that is **quantized** to a fixed set of values rather than a continuous range of values. A continuous value is quantized when it is approximated by a finite value, such as a bit pattern in a computer. Floating-point number roundoff error can be thought of as a kind of aliasing introduced by the quantization of real numbers to the floating-point represtnation; usually our calculations are such that the roundoff error intrinsic to a floating-point representation doesn't hamper the results of the calculation because it lies within acceptable error bounds. However, computer graphics often deals with values that have been quantized to relatively few bits of fixed-point precision – color channels can be quantized to anywhere from 1 to 8 bits of precision on Windows. Screen coordinates are typically quantized to 12 bits or less with 9 or 10 bits being the most common.

Antialiasing refers to the techniques used in computer graphics to minimize or eliminate aliasing. The mathematics of aliasing and its properties are very well understood, but are beyond the depth we will discuss it here. Anyone can appreciate the difference in visual quality between an aliased image and a properly antialiased image.

Direct3D provides a variety of antialiasing methods. Devices can provide full-scene antialiasing with multisampling, per-primitive antialiasing capabilities, texture antialiasing through mipmaps, 3D texture or volume antialiasing through mipvolumes, lighting antialiasing through multitexturing and depth antialiasing through w buffers.

1.11 Style Conventions

The following coding style conventions have been adopted throughout the example code used in this book for example code, Direct3D interfaces, macros, functions and types.

Macros

When describing C++ pre-processor macros, they will be listed as if they were inline functions with datatypes for input arguments and a return type where applicable. The macros themselves don't enforce this usage so these datatypes are given only as a guide to the reader as to the intended purpose of the macro.

Types

The Direct3D header files define an aliases for the Direct3D interface pointers. For instance, a pointer to the interface `IDirect3D8` has the typedef `LPDIRECT3D8`. Rather than use the typedef, the interface name will be used in the text. Mixed case names are easier to read and we wish to make it clear when a function takes a pointer to a pointer by using the `**` syntax. This book will also follow this convention for any conventional Win32 structures or intrinsic C++ datatypes.

Properties and Methods

A common schema for describing an object is to group its behavior into three categories: **properties** that represent the internal state of the object, **methods** or actions that can be taken on the object, and events or signals broadcast by the object to a group of subscribers. Direct3D objects expose no events, leaving us with properties and methods.

Direct3D properties are read with a `Get` method and written with a `Set` method on the interface. Properties may be read-only, write-only, or read-write depending on the methods provided in the interface. Property methods will be grouped by property name when an interface is summarized.

Methods are the interface methods that cause the object to do something interesting, like render a scene, and typically have verbs or verb noun phrases as their name. An application uses Direct3D objects by setting the appropriate object state through the property methods and then causing something interesting to happen through the action methods.

The members of an interface are listed in the sections of read-only properties, write-only properties, read/write properties and methods. Within each section, the relevant interface members are listed alphabetically, by property or method name. The SDK header files list interface members in "vtable" order, which is the required for proper operation, but makes for a poor reference.

Formal Argument Names

When formal argument names for API functions or COM object methods are given, the following conventions will be followed. When a set or get method is used to access a property, the argument corresponding to the property value will be named `value`. Similarly, when a method returns an interface pointer, the returned interface pointer argument will be named `value`. The remaining

arguments will be named according to their semantic role for the function or method.[6]

Structures and Unions

In C++, a structure is equivalent to a class with all members having public visibility. D3DX extension classes exploit this by inheriting from the Direct3D structures to extend them with member functions but no data. When the context is clear, structure member names are used directly, otherwise they are prefixed with a class scope prefix, such as `D3DCAPS8::DevCaps`.

Function and Class Names

When functions and classes are provided in example code they will be identified with lower-case names whose words are separated by underscores. This differentiates the example code in the book from GDI or other Win32 function calls at a glance. In addition, C++ standard library routines and classes will be prefixed with `std::` rather than importing the `std` namespace.

1.12 COM Objects

COM objects are **binary components** that are **servers** providing functionality to **clients** through **interfaces**. Direct3D is provided by Windows as a collection of objects that act as servers for graphic operations requested by your application as the client. Because COM provides a binary specification, the components may be written in any language that supports the creation of COM components. The COM runtime provides the infrastructure for locating specific binary components on your machine and attaching them to your application when you request their services. Binary components can be used from any programming environment that understands COM services, such as Visual Basic, Visual C++ or Delphi Object Pascal.

A way is needed to specify a particular component as well as the particular interface desired. COM uses **GUID**s to identify both components and interfaces. GUIDs are 128-bit numbers generated by the algorithm described in the Distributed Computing Environment specification and are guaranteed to be unique across time and space.[7]

Components present interfaces as an immutable contract between the server and the client. Once an interface is published to clients, it cannot be changed. If new functionality is needed or an interface needs to be changed, a new interface

[6]This differs from the SDK documentation. We prefer to give an independent naming over duplication of the SDK.

[7]The DCE algorithm uses the physical network address of the machine's netork adapter for uniqueness in space. These identifiers are guaranteed to be unique. If no network adapter is present when the GUID is generated, then the GUID is guaranteed only to be statistically unique.

must be created. This ensures backward compatability at the interface level with all existing clients of the component.

COM interfaces can be arranged in a hierarchy to extend existing interfaces, but only single inheritence of interfaces is allowed. This does not turn out to be a serious limitation because a COM object may provide any number of interfaces. COM implements polymorphism in this manner.

COM objects are reference counted. Every time a client obtains an interface on a COM object, the object's reference count is increased. Note that objects are reference counted and not interfaces; an object supporting multiple interfaces will have its reference count increased by one for each interface pointer obtained by a client. When the client is finished with the interface, it releases it and the object's reference count is decreased. When the object's reference count decreases to zero, the object may be safely destroyed. Just like memory allocated on the heap, COM objects can be "leaked" if the interfaces are not released. A leaked COM object should be treated like a memory leak and eliminated during application development.

All COM interfaces inherit from `IUnknown`, a distinguished base interface that manages the object's reference count and provides a means of obtaining any other interface supported by the object. The `IUnknown` interface is summarized in interface 1.1.

IUnknown

Methods	
`AddRef`	Add a reference to an object.
`QueryInterface`	Query an object for another interface pointer.
`Release`	Release a reference to an object.

Interface 1.1: Summary of the `IUnknown` interface.

```
interface IUnknown
{
 HRESULT QueryInterface(REFIID iid,
             void **result);
    UINT AddRef();
    UINT Release();
};
```

In C++, a COM object interface is presented to the application as a pure virtual base class with only public methods. It is impossible to instantiate a pure virtual base class, ensuring that an application can only obtain an interface pointer from the COM runtime or a COM object interface method such as `QueryInterface`. To invoke a method on a COM object, you obtain a pointer to an interface supported by the object and then invoke the method like you would any other C++ class instance pointer.

Predicates	
bool SUCCEEDED(HRESULT hr)	true when hr indicates success.
bool FAILED(HRESULT hr)	true when hr indicates failure.
Accessors	
DWORD HRESULT_FACILITY(HRESULT hr)	Extracts facility code.
DWORD HRESULT_CODE(HRESULT hr)	Extracts status code.

Table 1.5: Macros for HRESULT return values.

Many COM methods, such as QueryInterface return the type HRESULT. This is a DWORD sized quantity interpreted as three fields: a sucess bit, a facility code and a status code. The sucess bit indicates the failure or success of the operation as a whole. The facility code indicates the facility of origin in the system. The status code provides extended information besides that indicated in the success bit. Methods may succeed or fail with multiple distinct return codes. Use the macros in table 1.5 to examine HRESULT return values.

In Direct3D, when you pass an interface pointer to a COM object, it calls AddRef on the interface pointer if it stores the pointer as part of the object's internal state. No AddRef call is made if the object uses the interface only for the duration of its method. When a COM object replaces an existing interface pointer stored in its internal state with a new pointer, it first calls Release on the older interface pointer before overwriting it with the new interface pointer on which it calls AddRef. When an application wishes to force the release of an internally held interface, it instructs the COM object to store a null pointer into its internal storage, forcing a release of any internally held interface.

Before the COM runtime can be used, it must be initialized via the routine ::CoInitialize. This is typically done at startup. Similarly, an application calls ::CoUninitialize when finishing to free any resources allocated by the runtime. If an application does not use the COM runtime, then it needn't call either of these routines.

COM provides a standard object factory routine for creating instances (::CoCreateInstance, ::CoCreateInstanceEx). An object can also provide its own factory routine for use by an application. ::CoCreateInstance creates an instance and obtains a pointer to one of its interfaces. Since every interface inherits from IUnknown, you can always obtain the IUnknown interface pointer from any COM object instance. With IUnknown we can query for other interfaces and obtain an interface pointer if that interface is implemented by the object. ::CoCreateInstanceEx allows us to obtain multiple interfaces from a single object with one call. ::CoCreateInstanceEx is preferred for multithreaded applications for more control over the threading model used by COM objects.

Except for the initial IDirect3D8 interface obtained through ::Direct3DCreate8[8], all Direct3D COM objects are created by other Direct3D COM objects. A Direct3D application doesn't call ::CoCreateInstance for Direct3D COM objects and rarely calls QueryInterface because the Direct3D API is

[8] X file and D3DX objects also have factory functions.

streamlined for the operations typically needed by applications.

Because all Direct3D objects are obtained through factory functions and methods, Direct3D doesn't require the COM runtime to be initialized. If the application uses other COM objects or functions from the COM runtime, it will still need a call to `::CoInitialize` or `::CoInitializeEx`.

A simple COM application that plays `CLOCKTXT.AVI` from the SDK's media directory with DirectShow is given in listing 1.1. It performs error checking and explicitly manages the lifetime of the COM objects it uses with the `Release` method.

Listing 1.1: A simple AVI file player.

```
1   #define WIN32_LEAN_AND_MEAN
2   #define STRICT
3   #include <windows.h>
4   #include <dshow.h>
5
6   int APIENTRY
7   WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
8   {
9       HRESULT hr = ::CoInitialize(NULL);
10      if (FAILED(hr))
11      {
12          return -1;
13      }
14
15      // create the filter graph manager
16      IGraphBuilder *graph = 0;
17      hr = ::CoCreateInstance(CLSID_FilterGraph, NULL,
18          CLSCTX_ALL, IID_IGraphBuilder,
19          reinterpret_cast<void **>(&graph));
20      if (FAILED(hr))
21      {
22          ::CoUninitialize();
23          return -1;
24      }
25
26      // build the graph
27      hr = graph->RenderFile(L"CLOCKTXT.AVI", NULL);
28      if (FAILED(hr))
29      {
30          graph->Release();
31          ::CoUninitialize();
32          return -1;
33      }
34
```

```
35      // run the graph
36      {
37          IMediaControl *control = 0;
38          hr = graph->QueryInterface(IID_IMediaControl,
39              reinterpret_cast<void **>(&control));
40          if (FAILED(hr))
41          {
42              graph->Release();
43              ::CoUninitialize();
44              return -1;
45          }
46          hr = control->Run();
47          if (FAILED(hr))
48          {
49              control->Release();
50              graph->Release();
51              ::CoUninitialize();
52              return -1;
53          }
54          control->Release();
55      }
56
57      // wait for the AVI to complete
58      {
59          long event_code = 0;
60          IMediaEvent *event = 0;
61          hr = graph->QueryInterface(IID_IMediaEvent,
62              reinterpret_cast<void **>(&event));
63          if (FAILED(hr))
64          {
65              graph->Release();
66              ::CoUninitialize();
67              return -1;
68          }
69          hr = event->WaitForCompletion(INFINITE,
70              &event_code);
71          if (FAILED(hr))
72          {
73              event->Release();
74              graph->Release();
75              ::CoUninitialize();
76              return -1;
77          }
78          event->Release();
79      }
80
```

```
81      graph->Release();
82      ::CoUninitialize();
83
84      return 0;
85  }
```

1.13 Code Techniques

This section presents some code techniques you may find helpful during development of your Direct3D C++ application.

C++ Exceptions and `HRESULT`s

Almost every Direct3D interface method return `HRESULT` status codes. A well behaved program should check these values and act accordingly. Direct3D provides error checking on parameter values and additional error return codes when the debug run-time is in use. See chapter 19.

A typical COM program, such as the simple AVI player in listing 1.1, has to check many `HRESULT` return values. Every function is cluttered with the error checking code and the main path of execution isn't clear. Similarly, if we used `HRESULT`s in application code to propagate failure codes back up the call chain, all functions must deal with failure codes from called functions and propagate them back up to their caller.

```
HRESULT
DoSomething(ISomeInterface *iface)
{
    HRESULT hr = iface->Method1();
    if (FAILED(hr))
    {
        return hr;
    }
    hr = iface->Method2();
    if (FAILED(hr))
    {
        return hr;
    }
    hr = iface->Method3();
    if (FAILED(hr))
    {
        return hr;
    }
    return S_OK;
}
```

```
HRESULT
DoThings()
{
    ISomeInterface *iface = GetInterface();
    HRESULT hr = DoSomething(iface);
    if (FAILED(hr))
    {
        iface->Release();
        return hr;
    }
    hr = DoSomethingElse(iface);
    if (FAILED(hr))
    {
        iface->Release();
        return hr;
    }
    iface->Release();

    return S_OK;
}
```

Writing out if tests for every method can be tedious and error-prone. The readability of the program suffers as every function is littered with uncommon case error handling code and the commonly taken successful flow of control is obscured. Additionally as we allocate temporary resources through the course of the function, each must be released on an unexpected failure. Wether you repeat the cleanup code, as in listing 1.1, or use a `goto` with multiple labels for various amounts of cleanup at the end of the routine, the style is awkward and error-prone.

C++ exceptions and resource acquisition classes provide a mechanism for localizing the error handling code to one place and improving the readability of the main code path without giving up error detection. A resource acquisition class is a class that acquires some sort of resource in its constructor and releases the resource in its destructor. The resource can be temporarily allocated memory, COM interface pointers, mutual exclusion locks, open file handles, etc. Acquiring such resources with helper classes allocated on the stack ensures that the resources are properly released, even when an exception is thrown. When an exception is thrown, the call stack is unwound and any destructors for objects allocated on the stack are executed. When the call occurs normally, no exception is thrown and the helper object's destructor is called when the object goes out of scope. Either way, the resource is properly released when it is no longer needed.

To map `HRESULT`s to C++ exceptions, we need an exception class, an inline function and a preprocessor macro. First, a class `hr_message` is provided that encapsulates the error state of an `HRESULT`, a source code filename and a line number. Next, an inline function is provided that examines an `HRESULT` and

throws an exception of type `hr_message` if the `HRESULT` indicates failure. If the `HRESULT` indicates success, the inline function returns its `HRESULT` argument. Finally, a preprocessor macro is provided that automatically fills in the filename and line arguments when invoked. The complete header file is given in listing 1.2.

Listing 1.2: `<rt/hr.h>` mapping `HRESULT`s to exceptions.

```
1   #if !defined(RT_HR_H)
2   #define RT_HR_H
3   // hr.h
4   //
5   // Description: Handling unexpected COM HRESULTs as C++
6   // exceptions.
7   //
8   // The utilities provided here aid in debugging (and
9   // logging after delivery) as they record the source
10  // file/line location where the error was encountered.
11  //
12  // Setting breakpoints on the throw statemens in this
13  // header file will stop execution for all errors
14  // encountered by the program through THR(), TWS(), etc.,
15  // before the exception is thrown, allowing you to check
16  // the call stack for the source of the error in the
17  // debugger.
18  //
19  // This file is meant to be used to trap unexepected
20  // errors.  For expected error conditions (such as the
21  // HRESULTs returned by IDirect3D8::CheckDeviceType, or
22  // the D3DERR_DEVICELOST result returned by
23  // IDirect3DDevice8::Present), you should explicitly test
24  // against expected failure codes and only THR() on error
25  // code you do not expect.
26  //
27  // This will avoid the cost of exceptions for the normal
28  // flow of control, and the overhead of exceptions is
29  // perfectly reasonable when you expect the call to succeed
30  // but it fails anyway.  These failures usually represent
31  // an invalid parameter passed to Direct3D and the returned
32  // HRESULT will be D3DERR_INVALIDCALL with coresponding
33  // additional information in the debug output.
34  //
35  // Provides:
36  //  hr_message
37  //      Exception class that records a failed HRESULT,
38  //      file/line source file pair indicating where the
```

```
39  //      failed HRESULT was generated, and possible context
40  //      message.  Its constructor (not inlined) looks up
41  //      the HRESULT via FormatMessage() and a few other
42  //      places specific to DirectX to generate the message
43  //      text.
44  //
45  //  display_error
46  //      Displays a message box containing the error string
47  //      inside an hr_message and returns the HRESULT.
48  //
49  //  throw_hr, throw_win functions
50  //      Inline function for checking a result and throwing
51  //      an exception of type hr_message upon failure.
52  //
53  //  THR(), TWS() and variants
54  //      Macros to supply __FILE__ and __LINE__ at to
55  //      throw_hr/throw_win so that the file/line is
56  //      recorded from the source including this header and
57  //      not this header itself.
58  //
59  // Example:
60  //      try
61  //      {
62  //          THR(some_interface_ptr->SomeMethod());
63  //          HFONT font = TWS(::CreateFontIndirect(&lf));
64  //          // other stuff that may throw rt::hr_message
65  //      }
66  //      catch (const rt::hr_message &bang)
67  //      {
68  //          return rt::display_error(bang);
69  //      }
70  //
71  // Copyright (C) 2000-2001, Rich Thomson, all rights reserved.
72  //
73
74  #include <windows.h>
75  #include <tchar.h>
76
77  namespace rt
78  {
79      ///////////////////////////////////////////////////////////
80      // hr_message
81      //
82      // Class for bundling up an HRESULT and a message and a
83      // source code file/line number.
84      //
```

```
85      class hr_message
86      {
87      public:
88          hr_message(const TCHAR *file, unsigned line,
89                     HRESULT hr = E_FAIL,
90                     const TCHAR *message = NULL);
91          ~hr_message() {}
92
93          const TCHAR *file() const    { return m_file; }
94          unsigned line() const        { return m_line; }
95          HRESULT result() const       { return m_result; }
96          const TCHAR *message() const { return m_message; }
97
98      private:
99          enum
100         {
101             MESSAGE_LEN = 1024
102         };
103         const TCHAR *m_file;
104         unsigned m_line;
105         HRESULT m_result;
106         TCHAR m_message[MESSAGE_LEN];
107     };
108
109     /////////////////////////////////////////////////////////
110     // throw_hr
111     //
112     // Function that throws an exception when the given
113     // HRESULT failed.
114     //
115     inline HRESULT
116     throw_hr(const TCHAR *file, unsigned line,
117              HRESULT hr, const TCHAR *message = NULL)
118     {
119         if (FAILED(hr))
120         {
121             throw hr_message(file, line, hr, message);
122         }
123         return hr;
124     }
125
126     /////////////////////////////////////////////////////////
127     // throw_win
128     //
129     // Function that throws an exception when a Win32 return
130     // value indicates failure.
```

```
131     //
132     template<typename T>
133     inline T
134     throw_win(const TCHAR *file, unsigned line,
135               T status, const TCHAR *message = NULL,
136               int error = GetLastError())
137     {
138         if (!status)
139         {
140             throw_hr(file, line, HRESULT_FROM_WIN32(error),
141                      message);
142         }
143         return status;
144     }
145
146     /////////////////////////////////////////////////////////
147     // display_error
148     //
149     // Takes an hr_message and displays the message string in
150     // a message box and returns the HRESULT value.
151     //
152     inline HRESULT
153     display_error(const hr_message &bang,
154                   const TCHAR *title = NULL)
155     {
156         ::MessageBox(0, bang.message(), title, 0);
157         return bang.result();
158     }
159 };
160
161 // macros to fill in __FILE__, __LINE__ and _T() automatically
162
163 // THR => throw HRESULT
164 #define THR(hr_) \
165     rt::throw_hr(_T(__FILE__), __LINE__, hr_, _T(#hr_))
166 #define THRM(hr_, msg_) \
167     rt::throw_hr(_T(__FILE__), __LINE__, hr_, msg_)
168 #define THRMT(hr_, msg_) \
169     rt::throw_hr(_T(__FILE__), __LINE__, hr_, _T(msg_))
170
171
172 // Win32 has lots of functions that return zero on failure:
173 // a NULL pointer or handle, a zero return count, etc.
174 // Most Win32 functions return the error code via
175 // GetLastError().  Some Win32 functions return the error
176 // code as a non-zero status.  So throw_win takes both a
```

```
177  // status code and an error code.
178  //
179  // TWS   => throw Win32 function status
180  // TWSM  => Win32 status with message
181  // TWSMT => Win32 status with message constant needing _T()
182  #define TWS(status_) \
183      rt::throw_win(_T(__FILE__), __LINE__, status_)
184  #define TWSM(status_, msg_) \
185      rt::throw_win(_T(__FILE__), __LINE__, status_, msg_)
186  #define TWSMT(status_, msg_) \
187      rt::throw_win(_T(__FILE__), __LINE__, status_, _T(msg_))
188
189  // variations with error code supplied
190  #define TWSE(status_, error_) \
191  rt::throw_win(_T(__FILE__), __LINE__, status_, NULL, error_)
192  #define TWSME(status_, error_, msg_) \
193  rt::throw_win(_T(__FILE__), __LINE__, status_, msg_, error_)
194  #define TWSMTE(status_, err_, msg_) \
195  rt::throw_win(_T(__FILE__), __LINE__, status_, _T(msg_), err_)
196
197  #endif
```

Having localized the error handling with these tools, we can transform our hypothetical sample into the following:

```
class some_ptr
{
public:
    some_ptr(ISomeInterface *some) : m_some(some) {}
    ~some_ptr() { m_some->Release(); }
    operator ISomeInterface *() const { return m_some; }

private:
    ISomeInterface *m_some;
};

void
DoSomething(ISomeInterface *some)
{
    THR(some->Method1());
    THR(some->Method2());
    THR(some->Method3());
}

bool
```

```
DoThings()
{
    try
    {
        some_ptr some(GetInterface());
        DoSomething(some);
        DoSomethingElse(some);
    }
    catch (const rt::hr_message &bang)
    {
        // log unexpected error here
        return false;
    }

    return true;
}
```

This is admittedly a contrived example, but the main flow of control is now clearly visible, at the expense of wrapping each method in an invocation of the `THR` macro and the entire function in a try/catch block. Note also we had to wrap the `ISomeInterface` in a smart pointer so that it would be properly released in case `DoSomething` or `DoSomethingElse` threw an exception.

A more realistic example would place the try/catch at an outer scope, where error context information would be logged for debugging or displayed with a dialog box. This is especially true if `DoSomething` is called from a tight inner loop.

Another advantage of localizing the code for error handling is that a breakpoint can be set on the throw statement in `throw_hr`. When an error is encountered that would throw an exception, the breakpoint is activated and the programmer can examine the call stack and program state at the exact location of the error. The traditional if statement approach requires multiple breakpoints be set, or code single-stepped if the location of the error isn't known.

C++ exceptions are powerful and must be used with care. The main and most common flow of control should never be through a throw/catch pair as there is some overhead to exceptions. Like most programming features provided at runtime, excpetions come with a cost. When used reasonably, this cost should be insignificant compared to the time you save while debugging. Once your program is debugged, you can even redefine the `THR` macros so that exceptions cannot be thrown.

Whenever C++ exceptions are used, it is important to be aware of potential resource leaks. Objects created on the stack that are active at the time of a C++ exception have their destructors invoked as the stack frame is unwound from the stack frame of the throw statement to the stack frame of the enclosing catch statement. Dynamically allocated resources are not automatically destroyed when an exception is thrown, only objects allocated on the stack. An application using C++ exceptions in its implementation should not let exceptions unwind

the stack into the operating system, such as the caller of your windows or dialog procedure. If you implement COM objects with exception handling in the implementation you must catch the C++ exceptions in the interface methods to prevent C++ exceptions from unwinding the stack into the COM runtime.

The CD-ROM accompanying this book contains an implementation of `THR` and associated infrastructure for mapping HRESULTs to C++ exceptions.

`THR` Without Exceptions

It is also possible to use all the same mechanisms described above without using C++ exceptions. The DirectX SDK samples use a common application framework that relies on `HRESULT`s and `bool` return values from methods instead of exceptions for error handling.

The samples accompanying this book that are based on the SDK sample framework check all their COM method calls with a variation of `THR` that asserts `false` when a failed `HRESULT` is encountered. This is robust enough for a sample application that demonstrates a feature in the API, but not sufficient for a robust runtime error handling in a production application.

Smart Pointers

ATL 3.0 provides two helper template classes, `CComPtr<>` and `CComQIPtr<>` that simplify the management of COM interface pointers. `CComPtr<>` is used when you want to obtain an interface pointer via `::CoCreateInstance`. `CComQIPtr<>` is used when you want to obtain an interface pointer via `QueryInterface`. These classes are located in the `<atlbase.h>` header file in the ATL includes directory.

The C++ standard library helper class `std::auto_ptr`, declared in `<memory>`, can be used to avoid leaks of dynamically allocated memory. You should read and understand the implementation in `<memory>` for `auto_ptr`, or `<atlbase.h>` for `CComPtr<>` and `CComQIPtr<>`. Misunderstandings of the pointer ownership policies for smart pointer classes can lead to bugs just as difficult to track down as those that spurred the development of smart pointer classes in the first place.

The `std::auto_ptr` class can only be used with a single object as it uses the scalar memory allocator `new` instead of the array allocateor `new[]`. For a smart array pointer, see `<http://www.boost.org/>`.

You can also write your own smart pointer classes. For instance, the C++ standard library doesn't provide functions for reference-counted pointers to heap memory the way COM objects are reference counted. You can also incrementally extend existing smart pointer classes. The example code on the CD-ROM extends the ATL smart pointers to make them easier to use with Direct3D COM objects. You can also write your own smart resource helper classes for other dynamic resources like open file handles, critical sections, mutexes, GDI objects, etc.

Revised Simple AVI Player

We now return to our simple AVI player program and enhance it with full error checking and smart pointers to manage the lifetimes of objects. This program uses C++ exceptions and resource helper classes to initialize COM. The program is now robust against unexpected failed HRESULTs that may occur.

Listing 1.3: An improved simple AVI player.

```
1   #define WIN32_LEAN_AND_MEAN
2   #define STRICT
3   #include <windows.h>
4   #include <atlbase.h>
5   #include <dshow.h>
6   #include <rt/hr.h>
7
8   // acquire COM runtime as a resource
9   class com_runtime
10  {
11  public:
12      com_runtime()  { THR(::CoInitialize(NULL)); }
13      ~com_runtime() { ::CoUninitialize(); }
14  };
15
16  // extend CComQIPtr<T> to throw on no interface
17  template <typename T>
18  class com_qi_ptr : public CComQIPtr<T>
19  {
20  public:
21      com_qi_ptr(IUnknown *base) : CComQIPtr<T>(base)
22      {
23          if (!p) THR(E_NOINTERFACE);
24      }
25      ~com_qi_ptr() {}
26  };
27
28  int APIENTRY
29  WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
30  {
31      try
32      {
33          com_runtime com;
34
35          // Create the filter graph manager
36          CComPtr<IGraphBuilder> graph;
37          THR(graph.CoCreateInstance(CLSID_FilterGraph));
38
```

```
39            // Build the graph.
40            THR(graph->RenderFile(L"CLOCKTXT.AVI", NULL));
41
42            // Run the graph.
43            THR(com_qi_ptr<IMediaControl>(graph)->Run());
44
45            // Wait for completion.
46            long evCode = 0;
47            THR(com_qi_ptr<IMediaEvent>(graph)->
48                WaitForCompletion(INFINITE, &evCode));
49        }
50        catch (const rt::hr_message &bang)
51        {
52            return rt::display_error(bang);
53        }
54
55        return 0;
56    }
```

1.14 Previous Versions of Direct3D

Direct3D 8.0 provides significant ease of use improvements as well as some exciting new features compared to Direct3D 7. There are also some things possible in version 7 that are not possible with version 8.

The most significant change from previous versions is that DirectDraw has been eliminated as a separate interface and its features have been combined into the Direct3DDevice interface. This provides advanced features like alpha blending and arbitrary rotation to 2D applications.

At the API level, version 8 uses a different set of header files than version 7. The core datatypes no longer include C++ class member functions with `D3D_OVERLOADS` only the data members themselves. The member functions are now part of the D3DX classes which inherit from the core structures. Structures are no longer adorned with a dwSize member that must be properly initialized before using the structure. Callback functions have been eliminated completely.

Device enumeration and selection has been significantly simplified and the corresponding device enumeration features from D3DX eliminated. It is now possible to examine the capabality structure for a device without creating an instance of the device.

Presentation of rendered imagery for display has also been simplified. The code paths for presentation in windowed mode and full-screen mode have been made identical with slightly different arguments to the presentation methods. The `Blt` API has been eliminated and replaced with a simplified `CopyRects` method for a simple copying of pixels; the presentation uses of `Blt` have been replaced by the presentation methods.

New geometric primitives and new pipeline features are provided. The new primitives are provided as both core primitives and D3DX convenience "primitives". The core introduces expanded point and spline surface primitives. D3DX introduces text, progressive mesh and subdivision surface primitives.

The pipeline now offers programmable vertex shaders encompassing the vertex procesing stages of the fixed-function pipeline in version 7: vertex blending, transformation, and lighting. Programmable pixel shaders encompass the multitexturing stage of the fixed-function pipeline in version 7. An application can continue to use the fixed-function pipeline in version 8 if the flexibility of vertex and pixel shaders is not required.

Devices now offer multisampling support for full-scene antialiasing, motion blur and depth of field effects. Devices also offer cursor support. Volume and mipmapped volume texture support has been added.

Direct access to the primary surface is prohibited in Direct3D 8, although a rectangular copying proxy interface is provided. DirectShow cannot be used directly with the interfaces exposed in DirectX 8 Graphics.[9]

1.15 Further Reading

Gems *Graphics Gems*, edited by Andrew S. Glassner.
Provides many useful and practical algorithms for solving small problems in graphics. After five volumes, the book series has been replaced by the *Journal of Graphics Tools*. The book's source code is available at `<ftp://graphics.stanford.edu/pub/Graphics/GraphicsGems/>`.

Blinn *Jim Blinn's Corner: Dirty Pixels*, by Jim Blinn. This and another companion volume reprint Blinn's column from the journal *IEEE Computer Graphics & Applications*. The July, 1989 issue contained the column "Dirty Pixels" on gamma correction.

Box *Essential COM*, by Don Box.
Provides an excellent introduction to COM and an explanation of the technology from both client and server points of view.

Brockschmidt *Inside OLE, 2nd ed.*, by Kraig Brockschmidt.
One of the first books that explains all of COM (despite the title referring to OLE) both from a client and a server's point of view. The entire book is included in the MSDN Library CD-ROM and on-line at `<http://msdn.microsoft.com/library/default.asp>`.

ChandlerFötsch *Windows 2000 Graphics API Black Book*, Damon Chandler and Michael Fötsch. Contains a chapter on Image Color Management and device color profiles.

[9]You can always write your own DirectShow filter. The `Texture3D` DirectShow sample in the SDK shows how to play video to a texture.

Glassner *Principles of Digital Image Synthesis*, Andrew Glassner.
Comprehensive coverage of digital image synthesis in two volumes. More information about the human visual system, CRT display technology, and color spaces. Errata are located on-line at `<http://research.microsoft.com/glassner/work/projects/pdis/pdis.htm>`.

Hall *Illumination and Color in Computer Generated Imagery*, Roy Hall.
Extensive coverage of color and its use in computer synthetic imagery.

Stoustrup *C++ Programming Language*, 3rd edition, by Bjarne Stroustrup.
Explains C++ exceptions, the `std::auto_ptr<>` template class and helper classes for use with exceptions. Errata are located on-line at `<http://www.research.att.com/~bs/3rd.html>`.

Thorell *Using Computer Color Effectively*, by L. G. Thorell and W. J. Smith.
Illustrative guide to selecting colors for visual presentation based on perception in the human visual system as well as practical considerations.

JGT *Journal of Graphics Tools*, edited by Andrew S. Glassner.
See `<http://www.acm.org/jgt/>`.