Chapter 10

Rasterization and Shading

"Colors are the smiles of nature. When they are extremely smiling, and break forth into other beauty besides, they are her laughs, as in the flowers."

Leigh Hunt: *The Seer*, 1840

"Colors speak all languages."

Joseph Addison: *The Spectator*, June 27, 1712

10.1 Overview

With rasterization, we finally get to see how the geometry described to Direct3D gets turned into pixels on the screen. Direct3D uses a process called scanline rendering to produce pixels from primitives. The term scanline comes from the structure of a video monitor, where lines of pixels are scanned by the electron beam during display.

Once vertex processing has completed, the primitives have been transformed into screen space, where one unit equals the size of a pixel on the render target. Points, lines and triangles are processed into pixels through a set of rasterization rules. The rasterization rules define which pixels are generated by a primitive in a consistent fashion so that when two primitives meet at a coincident vertex, no gaps appear between primitives and that all pixels are generated exactly once for coincident but non-overlapping geometry.

Each pixel generated by rasterization will be associated with a depth value, an RGBA diffuse color, an RGB specular color, a fog factor, and one or more sets of texture coordinates. The values generated by rasterization are passed to subsequent stages of the graphics pipeline for per-pixel processing and then incorporation into the render target.

Before getting into the details of scanline rendering, we'll first take a look at the shading and filling possibilities provided by Direct3D. Shading refers to how Direct3D computes the colors for the pixels, while filling refers to how Direct3D decides to generate pixels for the interior of primitives. Then we'll look at how scanline rendering generates pixels for the primitives supplied by Direct3D.

10.2 Shading and Filling

Rasterization is also referred to as "shading" and "filling" as different shades of the vertex colors are used to fill the interior of primitives. The simplest method of shading and filling is to pick a constant color and fill the interior of a primitive with that color. For objects constructed from triangles, this leads to a faceted appearance when each triangle's color is chosen based on lighting, and leads to a solid color filled silhouette of the object when all the triangles are filled with the same color.

When the triangles approximate a smooth surface and each vertex contains the normal of the true surface, then each vertex can have a different color computed for it during lighting. We can shade the triangle using the colors at the vertices to compute colors for the interior[1]. RS Shade Mode defines how Direct3D computes the interior colors for a primitive and is value taken from the D3DSHADEMODE enumeration.

```
typedef enum _D3DSHADEMODE {
    D3DSHADE_FLAT    = 1,
    D3DSHADE_GOURAUD = 2,
    D3DSHADE_PHONG   = 3
} D3DSHADEMODE;
```

Flat shading computes a single color for the entire filled portion of the primitive. The color chosen is that associated with the first vertex in the primitive. If a triangle with vertices $\{A, B, C\}$ is specified, the vertex data used for flat shading is that associated with the vertex A. Similarly, for a line segment with vertices A, B, flat shading will use the color associated with vertex A. Flat shading does not provide a realistic rendering for smooth surfaces approximated by triangles.

Gouraud shading linearly interpolates between the vertices to produce the associated vertex data for points of the interior of the primitive. So if a line segment A, B has the color white associated with vertex A and black associated with vertex B, then a point midway between A and B will have the color that is 50% white, halfway between white and black. For a triangle, the points in the interior of the triangle have a color that is a linear combination of the colors at the vertices, depending on the point's distance from each vertex. Points along the edges of the triangle are colored similarly to a line, as a linear interpolation between the colors at the two vertices defining the edge.

[1] As we only have colors at the vertices of a triangle from lighting, the "interior" also includes the pixels along the edges of the triangle

Phong shading is not supported by Direct3D, but is described here for completeness. In chapter 8 we described how lighting was computed at the vertices to produce vertex colors, which are used by shading. Phong shading interpolates the normal across the triangle and computes the lighting at each pixel to produce "per-pixel lighting". This produces a more realistic shading of the surface, but is more expensive to compute as lighting is a complex computation. While Direct3D does not support Phong shading directly, it is still possible to perform per-pixel lighting using textures as we will see in chapter 11.

RS Fill Mode, with a value of type `D3DFILLMODE`, defines which portions of primitives are drawn and shaded. The point fill mode draws only the vertices of primitives. The wire frame fill mode draws the edges of primitives, including non-degenerate edges of degenerate triangles. The solid fill mode draws the entire interior of primitives, skipping degenerate triangles.

```
typedef enum _D3DFILLMODE {
    D3DFILL_POINT     = 1,
    D3DFILL_WIREFRAME = 2,
    D3DFILL_SOLID     = 3
} D3DFILLMODE;
```

We have discussed shading and filling as if only the diffuse color was involved in the interpolation. However, Direct3D can also interpolate other data associated with each vertex: the opacity (alpha), the specular color, the depth, the fog factor and the associated texture coordinates. Each of these values can be interpolated across the primitive by the device.

`D3DCAPS9::ShadeCaps` defines the shading capabilities of the device. All devices support flat shading with no interpolation of the associated vertex data. Each bit set in `ShadeCaps` defines which portion of the vertex color components are interpolated across the primitive. If texturing is supported by the device, then it interpolates texture coordinates across a primitive.

```
#define D3DPSHADECAPS_COLORGOURAUDRGB    0x00000008L
#define D3DPSHADECAPS_SPECULARGOURAUDRGB 0x00000200L
#define D3DPSHADECAPS_ALPHAGOURAUDBLEND  0x00004000L
#define D3DPSHADECAPS_FOGGOURAUD         0x00080000L
```

We saw in chapter 7 that a perspective projection introduces a depth distortion to primitives, causing primitives closer to the viewer to appear larger than those farther away. Perspective projections also introduce a distortion in the interpolation of vertex data. A device can compensate for this with additional work in the interpolation of vertex color components when the `D3DPRASTERCAPS_COLORPERSPECTIVE` bit of `RasterCaps` is set. Similarly, texture coordinates are interpolated with perspective correction if the `D3DPTEXTURECAPS_PERSPECTIVE` bit of `TextureCaps` is set.

```
#define D3DPRASTERCAPS_COLORPERSPECTIVE 0x00400000L
#define D3DPTEXTURECAPS_PERSPECTIVE     0x00000001L
```

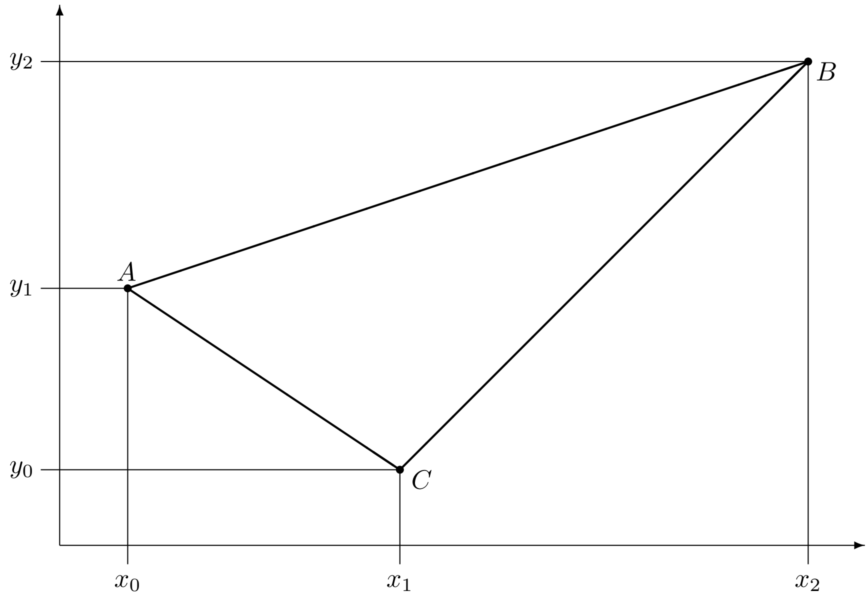

Figure 10.1: Scanline rendering of triangle ABC. The two dimensional points (x_0, y_1), (x_2, y_2) and (x_1, y_0) correspond to the vertices A, B, and C, respectively, after they are projected onto the screen.

10.3 Scanline Rendering

Scanline rendering is essentially a two-dimensional `for` loop over the pixel centers covered by a triangle. Before we discuss exactly what constitutes a "pixel center" or what "covering" means, let's take a look at the scanline rendering algorithm.

In figure 10.1 the triangle ABC is shown after it has been projected to screen space. This triangle has three edges AB, BC, and CA that form a boundary around the pixels we want to fill, assuming we're using a solid fill mode. We'd like to write our two-dimensional loop to iterate over the pixels inside the boundary formed by the edges, but the edges edges could be in any orientation on the screen. If we sort the edges based on the coordinates of their vertices, then we can write a doubly nested loop that iterates over scanlines inside the boundary and then the pixels within each scanline.

The loop will iterate from the bottom of the screen to the top of the screen for each scanline and then from the left side of the screen to the right side within each scanline. First, we sort the edges of the triangle based on the smallest y coordinate in each edge. This gives us an edge list $\{(x_1, y_0), (x_2, y_2)\}$, $\{(x_1, y_0), (x_0, y_1)\}$, $\{(x_0, y_1), (x_2, y_2)\}$.

Now we can process this list of edges in order of increasing y coordinate to generate scanlines between the edges. We loop over the scanlines, in order of increasing y, maintaining a list of active edges. Initially the active edge list is empty. Every time we loop on a scanline, we look at the list of edges sorted by y and move all edges that intersect this scanline to the active edge list. We keep the active edge list sorted by increasing x every time an edge is added to the list. With a list of edges that are active for the current scanline, we can loop through the edges, generating pixels between each pair of active edges. In the case of a single triangle, only 2 edge will be active at any time, but this algorithm scales up to rendering multiple triangles at the same time. At the end of each scanline, we increment the scanline counter y and remove any edges from the active edge list whose maximum y coordinate is smaller than the scanline counter.

For the triangle ABC, horizontal spans between the sides AC and BC will be produced, followed by spans between the sides AB and BC. The list of all edges and active edges will proceed as follows:

y	Edge List	Active Edge List
$0 \leq y < y_0$	$\{(x_1, y_0), (x_2, y_2)\}$, $\{(x_1, y_0), (x_0, y_1)\}$, $\{(x_0, y_1), (x_2, y_2)\}$	
$y_0 \leq y < y_1$	$\{(x_0, y_1), (x_2, y_2)\}$	$\{(x_1, y_0), (x_0, y_1)\}$, $\{(x_1, y_0), (x_2, y_2)\}$
$y_1 \leq y < y_2$		$\{(x_0, y_1), (x_2, y_2)\}$ $\{(x_1, y_0), (x_2, y_2)\}$
$y_2 \leq y$		

10.4 Source Pixel Generation

Now that we've seen how scanline rendering works, we can return to the question of what constitutes a "pixel center" and how a triangle is determined to "cover" a pixel center. In figure 10.2 we show the same triangle as in figure 10.1, showing the pixel centers and which centers are covered by the triangle. The pixels covered by a primitive are the source pixels for pixel processing.

For pixels in the interior of a triangle, they are clearly covered by the triangle and are filled as shown in the figure. The interesting case happens for pixels on or near the edge of the triangle. We want to select pixels for filling such that when two triangles coincide at their edges, each pixel is covered by exactly one and only one of the triangles. Direct3D accomplishes this by using a left-filling convention for determining which pixel centers are covered by a triangle. The left-filling convention means that each horizontal span of pixels generated through scanline rendering of a triangle is considered to be closed on the left and open on the right. In the figure, you can see that for the left side of each span, pixels whose centers are inside the triangle or on the edge of the triangle are filled by the left-filling convention. Pixels on the right side of each span are not considered inside the triangle if their pixel center coincides with the edge.

If another triangle shared the edge BC of the triangle shown, then the pixels along the edge would be filled by the other triangle. This ensures that the pixels along the boundary will be filled by exactly one triangle, even though the edge

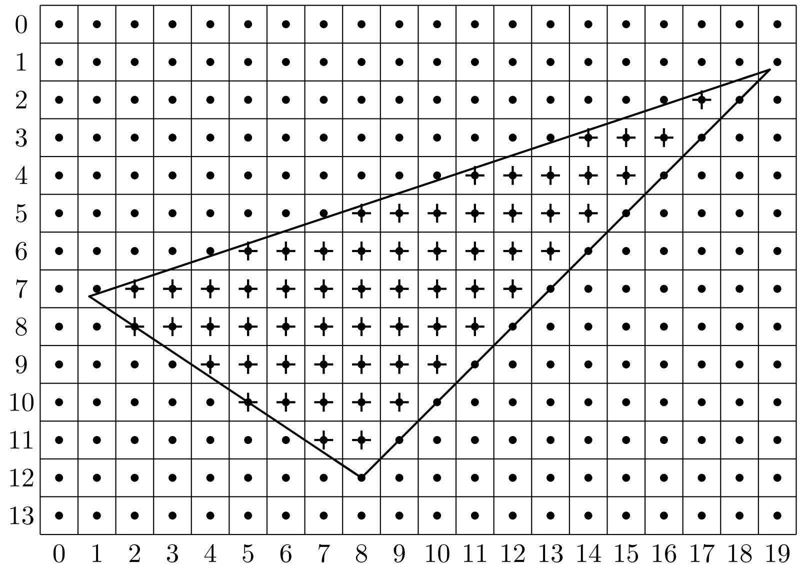

Figure 10.2: Rasterization of a triangle. Scanlines in screen space are numbered in screen space starting from 0 at the top of the screen, with y increasing downwards. Pixels in screen space are numbered from 0 at the left of the screen, with x increasing to the right. Pixel centers are shown with a solid dot and rasterized pixel centers are shown with a cross through the dot.

is shared. The vertex C is an interesting case because it is not filled even though you might think it should be filled. However, the pixel at $(8, 12)$ is not filled for this triangle because it is the right-most pixel in the span for that scanline. If another triangle shared the edge BC, then it the pixel $(8, 12)$ would be filled for that triangle.

Point Primitives

The rasterization of point and line primitives are similar to triangle primitives, but somewhat simplified because of the reduced dimensionality of these primitives. For points, the pixel center nearest to the projected coordinates of the point is considered covered by the point. Point sprites are conceptually rasterized as two textured triangles, although the exact method of rasterization is up to the device, following the rules for triangle rasterization.

Line Primitives

Line segments are rasterized in one of two ways, depending on their slope. For projected segments with a horizontal extent larger than their vertical extent, rasterization proceeds horizontally, where pixel centers vertically closest to the line are considered covered by the line. Similarly, for projected segments with a vertical extent larger than their horizontal extent, rasterization proceeds vertically where pixels centers horizontally closest to the line are considered covered by the line. In this manner, the pixels closest to a line are considered covered by the line, each covered pixel is generated only once, and the line never exceeds a single pixel in cross section.

There are times when it is convenient to have several lines coincide at a point, but have the coincident point rasterized only once. RS Last Pixel controls whether or not the last pixel of a rasterized line is generated. When `TRUE`, the last pixel is generated, otherwise the last pixel is not generated during rasterization. You can draw all but one of the coincident lines with RS Last Pixel disabled and draw the final coincident line with RS Last Pixel enabled to get the coincident point rasterized only once. The coincident vertex should be the second vertex in the line segment for this to work properly.

Lines can be antialiased with RS Antialiased Line Enable. When set to `TRUE`, all line primitives are rendered with an antialiasing filter, providing a smoother looking line. Patterning and antialiasing do not affect triangles drawn in wireframe mode, only line primitives. Using RS Antialiased Line Enable, we can use a two-pass technique to provide a smooth silhouette to a polygonal object. First, the polygonal object is drawn. Next, RS Antialiased Line Enable is set to `TRUE` and the silhouette of the object is drawn with line primitives. Care must be taken to provide the appropriate surface normal and color vertex components with the line primitives so that they match the solid object. As this involves drawing two primitives over top of the same pixels, care must be taken so that the proper visibility is applied with the Z buffer. See chapter 14 for more details on the Z buffer and visibility.

RS Antialiased Line Enable is supported if the `D3DLINECAPS_ANTIALIAS` bit of `D3DCAPS9::LineCaps` is set.

```
#define D3DLINECAPS_ANTIALIAS 0x00000020L
```

Many devices do not support line patterning or antialiasing, so alternative methods need to be found if these features are important. Lines can be patterned using textures, as we will see in chapter 11. Textures also provide a way to render triangles in such a manner that they appear as wide lines, or as antialiased lines. Multisampling can also be used to perform whole scene antialiasing, as described in chapter 14.

10.5 Source Pixel Data

Scanline rendering determines the x and y coordinates for each source pixel generated for a primitive. The z value for each vertex is also interpolated across the primitive to generate a depth value for each source pixel. Vertices may also have texture coordinates that are interpolated by the scanline rendering algorithm and associated with each generated source pixel. Texture coordinate processing is discussed in detail in chapter 11.

Diffuse and specular lighting vertex components are interpolated between vertices according to the value of RS Shade Mode. The diffuse lighting component's alpha channel determines the transparency of the vertex and is interpolated between vertices to generate the transparency for each source pixel. The specular lighting component's alpha channel contains the fog factor for each vertex and is interpolated between vertices to generate a fog factor for each source pixel. The associated information is passed to subsequent stages of the pipeline for pixel processing and incorporation into the frame buffer.

A device with a hardware rasterizer sets the `D3DDEVCAPS_HWRASTERIZATION`bit of `D3DCAPS9::DevCaps`. Cards without hardware rasterizers are extremely old and it is very unlikely you will encounter a card without this capability. This bit is informational in nature.

10.6 `rt_Rasterize` Sample Application

This sample application demonstrates the rasterization related render states RS Last Pixel, and RS Antialiased Line Enable.

The entire source code is included in the samples accompanying this book. Listed here is `rt_Rasterize.cpp`, containing the "interesting" code of the sample. The sample uses small helper classes that encapsulate reusable Direct3D coding idioms. Their meaning should be straightforward and all such helper classes are placed in the `rt` namespace to highlight their use.

Listing 10.1: `rt_Rasterize.cpp`: Demonstration of rasterization related render states.

```
//-----------------------------------------------------------------------------
// File: rt_Rasterize.cpp
//
// Desc: DirectX window application created by the DirectX AppWizard
//-----------------------------------------------------------------------------
#include <cmath>
#include <stdio.h>

#define STRICT
#include <windows.h>
#include <commctrl.h>
#include <commdlg.h>
#include <basetsd.h>

#include <atlbase.h>

#include <d3dx9.h>

#include "DXUtil.h"
#include "D3DEnumeration.h"
```

```
#include "D3DSettings.h"
#include "D3DApp.h"
#include "D3DFont.h"
#include "D3DUtil.h"

#include "resource.h"
#include "rt_Rasterize.h"

#include "vertices.h"

#include "rt/app.h"
#include "rt/colorsel.h"
#include "rt/hr.h"
#include "rt/misc.h"
#include "rt/states.h"
#include "rt/vertexbuf.h"

////////////////////////////////////////////////////////////
// s_circle_vertex::FVF
// s_grid_vertex::FVF
//
// FVF codes for the circle and grid vertices.
//
const DWORD
s_circle_vertex::FVF = D3DFVF_XYZ | D3DFVF_DIFFUSE;
const DWORD
s_grid_vertex::FVF = D3DFVF_XYZRHW | D3DFVF_DIFFUSE;

const UINT
CMyD3DApplication::NUM_CIRCLE_VERTICES = 200;

//-----------------------------------------------------------------------------
// Global access to the app (needed for the global WndProc())
//-----------------------------------------------------------------------------
CMyD3DApplication* g_pApp  = NULL;
HINSTANCE          g_hInst = NULL;




//-----------------------------------------------------------------------------
// Name: WinMain()
// Desc: Entry point to the program. Initializes everything, and goes into a
//       message-processing loop. Idle time is used to render the scene.
//-----------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, INT)
```

```
{
    CMyD3DApplication d3dApp;

    g_pApp  = &d3dApp;
    g_hInst = hInst;

    ::InitCommonControls();
    if (FAILED(d3dApp.Create(hInst)))
        return 0;

    return d3dApp.Run();
}




//-----------------------------------------------------------------------------
// Name: CMyD3DApplication()
// Desc: Application constructor.   Paired with ~CMyD3DApplication()
//       Member variables should be initialized to a known state here.
//       The application window has not yet been created and no Direct3D device
//       has been created, so any initialization that depends on a window or
//       Direct3D should be deferred to a later stage.
//-----------------------------------------------------------------------------
CMyD3DApplication::CMyD3DApplication() :
    CD3DApplication(),
    m_show_text(true),
    m_animate(false),
    m_last_pixel(true),
    m_antialias_lines(false),
    m_show_grid(false),
    m_show_circle(false),
    m_alpha_blending(true),
    m_grid_fg(D3DCOLOR_ARGB(100, 255, 255, 0)),
    m_circle_fg(D3DCOLOR_ARGB(100, 0, 255, 255)),
    m_background(D3DCOLOR_XRGB(0, 0, 0)),
    m_grid(),
    m_num_grid_segments(0),
    m_circle(),
    m_bLoadingApp(true),
    m_font(_T("Arial"), 12, D3DFONT_BOLD),
    m_input(),
    m_rot_x(0.0f),
    m_rot_y(0.0f)
{
    m_dwCreationWidth           = 500;
```

```
    m_dwCreationHeight          = 375;
    m_strWindowTitle            = TEXT("rt_Rasterize");
    m_d3dEnumeration.AppUsesDepthBuffer   = TRUE;
        m_bStartFullscreen                      = false;
        m_bShowCursorWhenFullscreen     = false;

    // Read settings from registry
    ReadSettings();
}




//-----------------------------------------------------------------------------
// Name: ~CMyD3DApplication()
// Desc: Application destructor.  Paired with CMyD3DApplication()
//-----------------------------------------------------------------------------
CMyD3DApplication::~CMyD3DApplication()
{
}




//-----------------------------------------------------------------------------
// Name: OneTimeSceneInit()
// Desc: Paired with FinalCleanup().
//       The window has been created and the IDirect3D9 interface has been
//       created, but the device has not been created yet.  Here you can
//       perform application-related initialization and cleanup that does
//       not depend on a device.
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::OneTimeSceneInit()
{
    // TODO: perform one time initialization

    // Drawing loading status message until app finishes loading
    ::SendMessage(m_hWnd, WM_PAINT, 0, 0);

    m_bLoadingApp = false;

    return S_OK;
}
```

```

//-----------------------------------------------------------------------------
// Name: ReadSettings()
// Desc: Read the app settings from the registry
//-----------------------------------------------------------------------------
VOID CMyD3DApplication::ReadSettings()
{
    HKEY hkey;
    if (ERROR_SUCCESS == ::RegCreateKeyEx(HKEY_CURRENT_USER, DXAPP_KEY,
        0, NULL, REG_OPTION_NON_VOLATILE, KEY_ALL_ACCESS, NULL, &hkey, NULL))
    {
        // TODO: change as needed

        // Read the stored window width/height.  This is just an example,
        // of how to use ::DXUtil_Read*() functions.
        ::DXUtil_ReadIntRegKey(hkey, TEXT("Width"), &m_dwCreationWidth, m_dwCrea
        ::DXUtil_ReadIntRegKey(hkey, TEXT("Height"), &m_dwCreationHeight, m_dwCr

        ::RegCloseKey(hkey);
    }
}




//-----------------------------------------------------------------------------
// Name: WriteSettings()
// Desc: Write the app settings to the registry
//-----------------------------------------------------------------------------
VOID CMyD3DApplication::WriteSettings()
{
    HKEY hkey;

    if (ERROR_SUCCESS == ::RegCreateKeyEx(HKEY_CURRENT_USER, DXAPP_KEY,
        0, NULL, REG_OPTION_NON_VOLATILE, KEY_ALL_ACCESS, NULL, &hkey, NULL))
    {
        // TODO: change as needed

        // Write the window width/height.  This is just an example,
        // of how to use ::DXUtil_Write*() functions.
        ::DXUtil_WriteIntRegKey(hkey, TEXT("Width"), m_rcWindowClient.right);
        ::DXUtil_WriteIntRegKey(hkey, TEXT("Height"), m_rcWindowClient.bottom);

        ::RegCloseKey(hkey);
    }
}
```

```
//-----------------------------------------------------------------------------
// Name: InitDeviceObjects()
// Desc: Paired with DeleteDeviceObjects()
//       The device has been created.  Resources that are not lost on
//       Reset() can be created here -- resources in D3DPOOL_MANAGED,
//       D3DPOOL_SCRATCH, or D3DPOOL_SYSTEMMEM.  Image surfaces created via
//       CreateImageSurface are never lost and can be created here.  Vertex
//       shaders and pixel shaders can also be created here as they are not
//       lost on Reset().
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::InitDeviceObjects()
{
    // Init the font
    m_font.InitDeviceObjects(m_pd3dDevice);

    return S_OK;
}




//-----------------------------------------------------------------------------
// Name: RestoreDeviceObjects()
// Desc: Paired with InvalidateDeviceObjects()
//       The device exists, but may have just been Reset().  Resources in
//       D3DPOOL_DEFAULT and any other device state that persists during
//       rendering should be set here.  Render states, matrices, textures,
//       etc., that don't change during rendering can be set once here to
//       avoid redundant state setting during Render() or FrameMove().
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::RestoreDeviceObjects()
{
    {
        HMENU menu = TWS(::GetMenu(m_hWnd));
        const bool enabled = (m_d3dCaps.LineCaps & D3DLINECAPS_ANTIALIAS) != 0;
        rt::enable_menu(menu, ID_OPTIONS_ANTIALIASLINES, enabled);
        if (!enabled && m_antialias_lines)
        {
            m_antialias_lines = false;
            rt::check_menu(menu, ID_OPTIONS_ANTIALIASLINES, false);
        }
```

```
    }

    // create grid vertices
    const UINT num_rows = (m_d3dsdBackBuffer.Height+15)/16;
    const UINT num_cols = (m_d3dsdBackBuffer.Width+15)/16;
    m_num_grid_segments = num_rows*(1 + num_cols);
    const UINT num_grid_vertices = m_num_grid_segments*2;
    THR(m_pd3dDevice->CreateVertexBuffer(sizeof(s_grid_vertex)*num_grid_vertices
        D3DUSAGE_WRITEONLY, s_grid_vertex::FVF, D3DPOOL_MANAGED, &m_grid, NULL))
    {
        rt::vertex_lock<s_grid_vertex> lock(m_grid);
        s_grid_vertex *vtx = lock.data();
        UINT i;
        for (i = 0; i < num_rows; i++)
        {
            *vtx = s_grid_vertex(0, i*16.f, m_grid_fg);
            vtx++;
            *vtx = s_grid_vertex(float(m_d3dsdBackBuffer.Width), i*16.f, m_grid_
            vtx++;
        }
        for (i = 0; i < num_rows; i++)
        {
            for (UINT j = 0; j < num_cols; j++)
            {
                *vtx = s_grid_vertex(j*16.f, i*16.f + 1, m_grid_fg);
                vtx++;
                *vtx = s_grid_vertex(j*16.f, (i+1)*16.f, m_grid_fg);
                vtx++;
            }
        }
    }

    // create circle VB
    THR(m_pd3dDevice->CreateVertexBuffer(sizeof(s_circle_vertex)*NUM_CIRCLE_VERT
        D3DUSAGE_WRITEONLY, s_circle_vertex::FVF, D3DPOOL_MANAGED, &m_circle, NU
    {
        rt::vertex_lock<s_circle_vertex> lock(m_circle);
        s_circle_vertex *vtx = lock.data();
        for (UINT i = 0; i < NUM_CIRCLE_VERTICES; i++)
        {
            vtx[i] = s_circle_vertex(i*D3DX_PI*2.f/(NUM_CIRCLE_VERTICES-1),
                m_circle_fg);
        }
    }

    // Set the world matrix
```

```
    D3DXMATRIX matIdentity;
    D3DXMatrixIdentity(&matIdentity);
    THR(m_pd3dDevice->SetTransform(D3DTS_WORLD,  &matIdentity));

    // Set up our view matrix. A view matrix can be defined given an eye point,
    // a point to lookat, and a direction for which way is up. Here, we set the
    // eye five units back along the z-axis and up three units, look at the
    // origin, and define "up" to be in the y-direction.
    D3DXMATRIX matView;
    D3DXVECTOR3 vFromPt   = D3DXVECTOR3(0.0f, 0.0f, -5.0f);
    D3DXVECTOR3 vLookatPt = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
    D3DXVECTOR3 vUpVec    = D3DXVECTOR3(0.0f, 1.0f, 0.0f);
    D3DXMatrixLookAtLH(&matView, &vFromPt, &vLookatPt, &vUpVec);
    THR(m_pd3dDevice->SetTransform(D3DTS_VIEW, &matView));

    // Set the projection matrix
    D3DXMATRIX matProj;
    FLOAT fAspect = ((FLOAT)m_d3dsdBackBuffer.Width) / m_d3dsdBackBuffer.Height;
    D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI/4, fAspect, 1.0f, 100.0f);
    THR(m_pd3dDevice->SetTransform(D3DTS_PROJECTION, &matProj));

    // Restore the font
    m_font.RestoreDeviceObjects();

    return S_OK;
}




//-----------------------------------------------------------------------------
// Name: FrameMove()
// Desc: Called once per frame, the call is the entry point for animating
//       the scene.
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::FrameMove()
{
    // Update user input state
    UpdateInput();

    // Update the world state according to user input
    D3DXMATRIX matWorld;
    D3DXMATRIX matRotY;
    D3DXMATRIX matRotX;

    if (m_animate || (m_input.m_left && !m_input.m_right))
```

```
        m_rot_y += m_fElapsedTime;
    else if (m_input.m_right && !m_input.m_left)
        m_rot_y -= m_fElapsedTime;

    if (m_animate || (m_input.m_up && !m_input.m_down))
        m_rot_x += m_fElapsedTime;
    else if (m_input.m_down && !m_input.m_up)
        m_rot_x -= m_fElapsedTime;

    m_rot_x = std::fmodf(m_rot_x, 2.f*D3DX_PI);
    m_rot_y = std::fmodf(m_rot_y, 2.f*D3DX_PI);

    ::D3DXMatrixRotationX(&matRotX, m_rot_x);
    ::D3DXMatrixRotationY(&matRotY, m_rot_y);

    ::D3DXMatrixMultiply(&matWorld, &matRotX, &matRotY);
    THR(m_pd3dDevice->SetTransform(D3DTS_WORLD, &matWorld));

    return S_OK;
}




//-----------------------------------------------------------------------------
// Name: UpdateInput()
// Desc: Update the user input.  Called once per frame
//-----------------------------------------------------------------------------
void CMyD3DApplication::UpdateInput()
{
    m_input.m_up    = (m_bActive && (GetAsyncKeyState(VK_UP)    & 0x8000) == 0x8
    m_input.m_down  = (m_bActive && (GetAsyncKeyState(VK_DOWN)  & 0x8000) == 0x8
    m_input.m_left  = (m_bActive && (GetAsyncKeyState(VK_LEFT)  & 0x8000) == 0x8
    m_input.m_right = (m_bActive && (GetAsyncKeyState(VK_RIGHT) & 0x8000) == 0x8
}




//-----------------------------------------------------------------------------
// Name: Render()
// Desc: Called once per frame, the call is the entry point for 3d
//       rendering. This function sets up render states, clears the
//       viewport, and renders the scene.
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::Render()
```

```
{
    // Clear the viewport: no Z buffer is used
    THR(m_pd3dDevice->Clear(0L, NULL, D3DCLEAR_TARGET,
        m_background, 1.0f, 0L));

    THR(m_pd3dDevice->BeginScene());

    // set rasterization state for lines:
    // RS Edge Antialias based on GUI state
    // RS Last Pixel based on GUI state
    // RS Line Pattern to default of 0 for grid
    // alpha blending,
    // no lighting or Z buffering
    rt::s_rs states[] =
    {
        D3DRS_LASTPIXEL, m_last_pixel,
        D3DRS_ANTIALIASEDLINEENABLE, m_antialias_lines,
        D3DRS_ALPHABLENDENABLE, m_alpha_blending,
        D3DRS_SRCBLEND, D3DBLEND_SRCALPHA,
        D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA,
        D3DRS_ZENABLE, D3DZB_FALSE,
        D3DRS_LIGHTING, FALSE
    };
    rt::set_states(m_pd3dDevice, states, NUM_OF(states));

    // draw the screen space grid if requested
    if (m_show_grid)
    {
        THR(m_pd3dDevice->SetFVF(s_grid_vertex::FVF));
        THR(m_pd3dDevice->SetStreamSource(0, m_grid, 0,
            sizeof(s_grid_vertex)));
        THR(m_pd3dDevice->DrawPrimitive(D3DPT_LINELIST, 0,
            m_num_grid_segments));
    }

    // draw world space circle if requested
    if (m_show_circle)
    {
        // set the line pattern for the circle
        THR(m_pd3dDevice->SetFVF(s_circle_vertex::FVF));
        THR(m_pd3dDevice->SetStreamSource(0, m_circle, 0,
            sizeof(s_circle_vertex)));
        THR(m_pd3dDevice->DrawPrimitive(D3DPT_LINESTRIP, 0,
            NUM_CIRCLE_VERTICES-1));
    }

```

```
    // Render stats and help text
    if (m_show_text)
    {
        RenderText();
    }

    THR(m_pd3dDevice->EndScene());

    return S_OK;
}




//-----------------------------------------------------------------------------
// Name: RenderText()
// Desc: Renders stats and help text to the scene.
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::RenderText()
{
    D3DCOLOR fontColor        = D3DCOLOR_ARGB(255,255,255,0);
    TCHAR szMsg[MAX_PATH] = TEXT("");

    // Output display stats
    float fNextLine = 40.0f;

    _tcscpy(szMsg, m_strDeviceStats);
    fNextLine -= 20.0f;
    m_font.DrawText(2, fNextLine, fontColor, szMsg);

    _tcscpy(szMsg, m_strFrameStats);
    fNextLine -= 20.0f;
    m_font.DrawText(2, fNextLine, fontColor, szMsg);

    // Output statistics & help
    fNextLine = (float) m_d3dsdBackBuffer.Height;
    _stprintf(szMsg, TEXT("Arrow keys: Up=%d Down=%d Left=%d Right=%d"),
              m_input.m_up, m_input.m_down, m_input.m_left, m_input.m_right);
    fNextLine -= 20.0f; m_font.DrawText(2, fNextLine, fontColor, szMsg);
    _stprintf(szMsg, TEXT("World State: %0.3f, %0.3f"),
                    m_rot_x, m_rot_y);
    fNextLine -= 20.0f; m_font.DrawText(2, fNextLine, fontColor, szMsg);
    _tcscpy(szMsg, TEXT("Use arrow keys to update input"));
    fNextLine -= 20.0f; m_font.DrawText(2, fNextLine, fontColor, szMsg);
    _tcscpy(szMsg, TEXT("Press 'F2' to configure display"));
    fNextLine -= 20.0f; m_font.DrawText(2, fNextLine, fontColor, szMsg);
```

```
    return S_OK;
}




//-----------------------------------------------------------------------------
// Name: MsgProc()
// Desc: Overrrides the main WndProc, so the sample can do custom message
//       handling (e.g. processing mouse, keyboard, or menu commands).
//-----------------------------------------------------------------------------
LRESULT CMyD3DApplication::MsgProc(HWND hWnd, UINT msg, WPARAM wParam,
                                    LPARAM lParam)
{
    LRESULT result = 0;
    bool handled = false;

    switch (msg)
    {
    case WM_PAINT:
        if (m_bLoadingApp)
        {
            // Draw on the window tell the user that the app is loading
            // TODO: change as needed
            HDC hDC = TWS(::GetDC(hWnd));
            RECT rct;
            TWS(::GetClientRect(hWnd, &rct));
            ::DrawText(hDC, TEXT("Loading... Please wait"), -1, &rct,
                DT_CENTER|DT_VCENTER|DT_SINGLELINE);
            TWS(::ReleaseDC(hWnd, hDC));
        }
        break;

    case WM_COMMAND:
        result = on_command(hWnd, wParam, lParam, handled);
        break;
    }

    return handled ? result : CD3DApplication::MsgProc(hWnd, msg, wParam, lParam);
}




//-----------------------------------------------------------------------------
// Name: InvalidateDeviceObjects()
```

```
// Desc: Invalidates device objects.  Paired with RestoreDeviceObjects()
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::InvalidateDeviceObjects()
{
    m_grid = 0;
    m_circle = 0;

    m_font.InvalidateDeviceObjects();

    return S_OK;
}




//-----------------------------------------------------------------------------
// Name: DeleteDeviceObjects()
// Desc: Paired with InitDeviceObjects()
//       Called when the app is exiting, or the device is being changed,
//       this function deletes any device dependent objects.
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::DeleteDeviceObjects()
{
    // TODO: Cleanup any objects created in InitDeviceObjects()
    m_font.DeleteDeviceObjects();

    return S_OK;
}




//-----------------------------------------------------------------------------
// Name: FinalCleanup()
// Desc: Paired with OneTimeSceneInit()
//       Called before the app exits, this function gives the app the chance
//       to cleanup after itself.
//-----------------------------------------------------------------------------
HRESULT CMyD3DApplication::FinalCleanup()
{
    // TODO: Perform any final cleanup needed

    // Write the settings to the registry
    WriteSettings();

    return S_OK;
```

```
}




///////////////////////////////////////////////////////////
// CMyD3DApplication::on_command
//
// WM_COMMAND handler -- update internal state based on GUI
// interaction.
//
LRESULT
CMyD3DApplication::on_command(HWND window, WPARAM wp, LPARAM, bool &handled)
{
    LRESULT result = 0;
    handled = false;
    const HMENU menu = ::GetMenu(window);
    const UINT control = LOWORD(wp);

    switch (control)
    {
#define TOGGLE(id_, state_)                 \
    case id_:                               \
        rt::toggle_menu(menu, id_, state_); \
        handled = true;                     \
        break
    TOGGLE(IDM_OPTION_SHOW_TEXT, m_show_text);
    TOGGLE(IDM_OPTION_ANIMATE, m_animate);
    TOGGLE(IDM_OPTION_LAST_PIXEL, m_last_pixel);
    TOGGLE(IDM_SCENE_GRID, m_show_grid);
    TOGGLE(IDM_SCENE_CIRCLE, m_show_circle);
    TOGGLE(ID_OPTIONS_ANTIALIASLINES, m_antialias_lines);
    TOGGLE(ID_OPTIONS_ALPHABLENDING, m_alpha_blending);
#undef TOGGLE

    case IDM_OPTION_BACKGROUND:
        {
            rt::pauser block(*this);
            m_background = rt::choose_color(window, m_background);
            handled = true;
        }
        break;

    case IDM_OPTION_GRID_COLOR:
        {
            rt::pauser block(*this);
```

```
            m_grid_fg = rt::choose_color(window, m_grid_fg);
            update_grid_color();
            handled = true;
        }
        break;

    case IDM_OPTION_CIRCLE_COLOR:
        {
            rt::pauser block(*this);
            m_circle_fg = rt::choose_color(window, m_circle_fg);
            update_circle_color();
            handled = true;
        }
        break;

    case IDM_OPTION_RESET_VIEW:
        m_rot_x = 0;
        m_rot_y = 0;
        break;

    default:
        // all our control IDs are > 40006 and we should handle them all
        ATLASSERT(control <= 40006);
    }

    return result;
}

////////////////////////////////////////////////////////////
// update_grid_color
//
// Store the new grid color in the grid vertices.
//
void
CMyD3DApplication::update_grid_color()
{
    rt::vertex_lock<s_grid_vertex> lock(m_grid);
    s_grid_vertex *vtx = lock.data();
    for (UINT i = 0; i < m_num_grid_segments*2; i++)
    {
        vtx[i].m_fg = m_grid_fg;
    }
}

////////////////////////////////////////////////////////////
// update_circle_color
```

```
//
// Store the new circle color in the circle vertices.
//
void
CMyD3DApplication::update_circle_color()
{
    rt::vertex_lock<s_circle_vertex> lock(m_circle);
    s_circle_vertex *vtx = lock.data();
    for (UINT i = 0; i < NUM_CIRCLE_VERTICES; i++)
    {
        vtx[i].m_fg = m_circle_fg;
    }
}
```