Chapter 14

The Frame Buffer

“A picture is a mute poem.
(Mutum est pictura poema.)”
Latin proverb

14.1 Overview

Rasterization generates a stream of source pixels from graphic primitives, which are combined with destination pixels in the frame buffer. The term frame buffer originates from the early days of raster graphics and referred to a bank of memory that contained a single image, or frame. As computer graphics evolved, the term came to encompass the image data as well as any ancillary data needed during graphic rendering. In Direct3D, the frame buffer encompasses the currently selected render target surface and depth/stencil surfaces. If multisampling is used, additional memory is required but is not explicitly exposed as directly manipulable surfaces.

After rasterization, each source pixel contains an RGB color, an associated transparency value in its alpha channel, and an associated depth in the scene in its Z value. The Z value is a fixed-point precision quantity produced by rasterization. Fog may then be applied to the pixel before it is incorporated into the render target. The application of fog mixes the pixel’s color value, but not its alpha value, from rasterization with a fog color based on a function of the pixel’s depth in the scene. Fog, also referred to as depth cueing, can be used to diminish the intensity of an object as it recedes from the camera, placing more emphasis on objects closer to the viewer.

After fog application, pixels can be rejected on the basis of their transparency, their depth in the scene, or by stenciling operations. Stencil operations allow arbitrary regions of the frame buffer to be masked away from rendering, among other things. Unlike the associated alpha and depth produced for each pixel during rasterization, the stencil value associated with the source pixel is obtained from a render state.
If the pixel passes the alpha, depth and stencil tests, then it will be combined into the render target. The source pixel’s depth value is used only for visibility computations and no further processing involving the depth value is performed after the depth test is applied. The pixel’s alpha value is used to combine the pixel into the render target through a formula that can mix the pixels from primitives with the pixels already present in the render target.

Finally, the source pixel may be processed through dithering to eliminate any banding artifacts when the render target contains a reduced dynamic range for the RGB colors. After dithering, the source pixel is ready to be written into the render target and associated depth/stencil surfaces. The write operations can be controlled through the use of write masks.

Multisampling provides multiple color, depth, and stencil values for each pixel in the render target. The multiple color values are combined during presentation to produce a final image that is used for video scan out. The additional depth and stencil values are required to obtain the proper visibility and stenciling effects with multisampling. Multisampling can provide antialiasing, depth of field, motion blur, and other effects. During rendering, you can control which samples of a multisample render target are used for the destination. Some effects will render to all of the samples associated with a pixel in a single pass, while other effects will render to a portion of the samples in each pass of a multipass technique.

14.2 Fog Blending

The fog color is blended into the source pixel’s RGB color just before any frame buffer processing occurs. The fog color affects only the pixel’s color and not its transparency. Fog is fully described in section 6.8.

14.3 Alpha Test

After the fog color has been applied to source pixels, they can be subjected to a rejection test based on the transparency (alpha) value of the source pixel. If the alpha test is enabled, a comparison is made between the alpha value of the source pixel and a fixed alpha value given by a render state. If the test fails, the pixel is discarded and no further processing is performed on the source pixel.

You can reject completely transparent pixels which would have no effect on the final rendering. This eliminates work in the frame buffer and can increase your rendering throughput when a significant number of pixels are likely to be completely transparent. This is often the case when a texture is used as a cut-out for a shape, with a large number of completely transparent pixels in the texture.

The alpha test comparison is given by the equation

\[ \alpha_s \langle op \rangle \alpha_r \]
14.3. ALPHA TEST

where $\alpha_s$ is the alpha value of the source pixel, $\alpha_r$ is the reference alpha value and $\langle op \rangle$ is the comparison function. Be careful not to confuse the alpha test, which rejects source pixels based on their alpha value, with alpha blending, which uses the alpha of the source pixel to combine the source pixel with the frame buffer.

RS Alpha Blend Enable determines if the alpha test is applied. If enabled, RS Alpha Func specifies the comparison function used to compare the reference value with the source pixel’s alpha value. The function is specified with a value from the D3DCMPFUNC enumerated type. RS Alpha Ref provides the reference alpha value used in the comparison. The reference value is a DWORD in the interval [0, 255], where 0 corresponds to fully-transparent and 255 corresponds to fully-opaque.

typedef enum _D3DCMPFUNC {
    D3DCMP_NEVER = 1,
    D3DCMP_LESS = 2,
    D3DCMP_EQUAL = 3,
    D3DCMP_LESSEQUAL = 4,
    D3DCMP_GREATER = 5,
    D3DCMP_NOTEQUAL = 6,
    D3DCMP_GREATEREQUAL = 7,
    D3DCMP_ALWAYS = 8
} D3DCMPFUNC;

The D3DCMPFUNC values correspond to the usual mathematical comparison operators, with two exceptions. The D3DCMP_NEVER and D3DCMP_ALWAYS values correspond to comparison functions which never succeed or always succeed, respectively.

D3DCAPS9::AlphaCmpCaps describes the alpha comparison functions supported by the device for area primitives (triangles, point sprites and higher-order surfaces). Each bit that is set corresponds to a supported comparison function.

#define D3DPMCPCAPS_NEVER 0x00000001L
#define D3DPMCPCAPS_LESS 0x00000002L
#define D3DPMCPCAPS_EQUAL 0x00000004L
#define D3DPMCPCAPS_LESSEQUAL 0x00000008L
#define D3DPMCPCAPS_GREATER 0x00000010L
#define D3DPMCPCAPS_NOTEQUAL 0x00000020L
#define D3DPMCPCAPS_GREATEREQUAL 0x00000040L
#define D3DPMCPCAPS_ALWAYS 0x00000080L

If the D3DLINECAPS_ALPHACMP bit of LineCaps is set, then the device supports alpha test comparisons for point and line primitives. The supported comparison functions are the same as for area primitives.

#define D3DLINECAPS_ALPHACMP 0x00000008L
14.4 The Z Buffer and Visibility

Visibility in Direct3D is usually determined by the Z buffer. There are other visibility algorithms, such as the painter’s algorithm, but the Z buffer is easy to implement in hardware and is the most commonly used visibility algorithm.

You can create a depth/stencil buffer for use with the device when the device is created by setting the AutoDepthStencil and AutoDepthStencilFormat members of the present parameters passed to CreateDevice. If you create additional render targets with CreateRenderTarget or additional swap chains, you may need to create depth/stencil buffers for use with those render targets. You can create a depth/stencil surface with CreateDepthStencilSurface for use with SetRenderTarget. GetDepthStencilSurface returns the depth/stencil surface associated with the current render target.

```c
HRESULT CreateDepthStencilSurface(UINT width, 
                                  UINT height, 
                                  D3DFORMAT format, 
                                  D3DMULTISAMPLE_TYPE multisample, 
                                  IDirect3DSurface9 **result);
HRESULT GetDepthStencilSurface(IDirect3DSurface9 **result);
```

The width and height of the depth/stencil surface should match the render target surface with which it will be used. For multisampling render targets, the multisample type of the depth/stencil surface should also match the render target. Multisampling is discussed in section 14.8.

The Z buffer visibility algorithm works by keeping a depth value for each pixel in the frame buffer. Before the scene is rendered, the Z buffer is initialized with the value of the farthest possible Z value. As source pixels are produced by rasterization, each pixel will have an associated Z value that is determined by the vertices of a primitive. The source pixel’s Z value is compared with the Z value stored for the destination pixel. If the source pixel’s Z value is closer to the viewer than the one already stored in the Z buffer, then the source pixel is closer than the destination pixel and it’s Z value is stored into the Z buffer at the destination pixel’s location. Successive source pixels then test themselves against the new Z value and only overwrite this value if they are even closer to the viewer. This amounts to a per-pixel sort for each source pixel that touches the same destination location in the render target. The visibility problem is essentially a sorting problem. This is not the only way the Z buffer can be used, but it is the most common.

The Z buffer test is enabled with RS Z Enable and has a value from the D3DZBUFFERTYPE enumeration. Z buffering is selected when the value D3DZB_TRUE is used. W buffering is selected with the value D3DZB_USEW. W buffering is explained later in this section.

```c
typedef enum _D3DZBUFFERTYPE {
    D3DZB_FALSE = 0,
    D3DZB_TRUE = 1,

```
When enabled, the Z buffer test is performed by the following equation:

\[ z_s \langle \text{op} \rangle z_d \]

where \( z_s \) is the Z value associated with the source pixel, \( z_d \) is the Z value stored in the depth buffer of the render target and \( \langle \text{op} \rangle \) is the comparison operator used. RS Z Func defines the comparison function \( \langle \text{op} \rangle \) and is given by a member of the D3DCMPFUNC enumeration.

When the D3DRASTERCAPS_ZTEST bit of D3DCAPS9::RasterCaps is set, the device supports the depth buffer test. D3DCAPS9::ZCmpCaps describes the Z buffer comparison functions supported by the device for area primitives (triangles, point sprites and higher-order surfaces). Each bit that is set corresponds to a supported comparison function. The supported functions are given by the D3DPCMPFUNC flags described on page 493. If the D3DLINECAPS_ZTEST bit of D3DCAPS9::LineCaps is set, the device supports depth test comparisons for point and line primitives.

```c
#define D3DRASTERCAPS_ZTEST 0x00000010L
#define D3DLINECAPS_ZTEST 0x00000002L
```

**W Buffering**

Low bit depths for the Z buffer can be a problem when a perspective projection transformation is used. Perspective foreshortening causes most of the values in the Z buffer to be used for the portion of the scene closest to the camera. This can result in visibility artifacts for primitives far away from the camera. The problem can be reduced by moving the near and far planes of the view frustum as close together as possible and moving the near plane as far away from the camera as possible. However, this does not eliminate the problem, it only lessens its severity.

An alternative to using a Z buffer is to use a W buffer, where the depth stored in the buffer is the reciprocal homogeneous depth value \( 1/w \). In this case, the values in the W buffer will be evenly spread across the depth of the view frustum. For a depth buffer with 16 bits of resolution, a W buffer can improve the resolution of visibility in the scene.

To use a W buffer, specify D3DZB_USEW for RS Z Enable. W buffering is supported by a device if the D3DRASTERCAPS_WBUFFER bit of D3DCAPS9::RasterCaps is set. A device supports W based fog if the D3DRASTERCAPS_WFOG bit of D3DCAPS9::RasterCaps is set. The MaxVertexW member of D3DCAPS9 gives the maximum W value for a vertex supported by the device.

```c
#define D3DRASTERCAPS_WBUFFER 0x00040000L
#define D3DRASTERCAPS_WFOG 0x00100000L
```
Visibility of Transparent Primitives

Because the Z buffer visibility algorithm is equivalent to a sorting algorithm that finds the pixel closest to the viewer, you can normally ignore the order in which primitives are drawn. When the primitives are fully opaque, only the pixels from the closest primitive will appear in the final rendering.

However, when the primitives are partially transparent, then whatever is behind the partially transparent primitives – hereafter referred to simply as “transparent primitives” for brevity – will show through them. To properly render transparent primitives they should be drawn in back-to-front order so that the primitives behind are properly obscured by the primitives in front. To render this sort of scene perfectly, we would need to keep a list of all transparent pixel colors and their associated depths for each destination pixel during rendering. When all the primitives have been rendered, we can collapse each list by sorting the pixels by their depth in the scene and combining them together with the appropriate blending. While this perfect rendering cannot be obtained with a simple Z buffer, which only stores a single depth value for each destination pixel, it is possible to achieve an approximate rendering with the Z buffer that reduces the error compared to drawing primitives in an arbitrary order.

First, all the completely opaque objects are drawn with the standard Z buffer visibility algorithm: the Z buffer is cleared to the value farthest from the camera and all the opaque objects in the scene are rendered and their Z values are written into the Z buffer. Next, the transparent objects in the scene are sorted in back-to-front order based on their bounding boxes. Then, writes to the Z buffer are disabled and the transparent objects are drawn in back-to-front order. By disabling writes to the Z buffer, but keeping the Z test enabled, the transparent objects are properly occluded by the opaque objects in the scene, but they may still interpenetrate other transparent objects. By drawing the transparent objects in back-to-front order, they properly obscure other transparent objects as long as the objects don’t interpenetrate. For perfect rendering of interpenetrating transparent objects, you would have to sort the individual triangles within the interpenetrating objects to ensure the proper back-to-front rendering order of the individual triangles. However, the additional visual quality obtained from this “perfect” rendering is usually not worth the additional computational cost in an interactive graphics program.

Biasing Primitives

Occasionally you will want to draw two primitives that are mathematically in the same location. For instance, you might want to draw a triangle on the side of a cube, where the cube is drawn with a separate primitive. You could approach this as a modeling problem and join the overlaid triangle and the side of the cube together. A simpler approach using the Z buffer is to add a fixed Z bias to the triangle to bring it “in front” of the cube.

RSZBias controls the amount of biasing added to area primitives. The bias can be a value in the range [0, 16], with larger values adding more bias to bring
the primitives “closer” to the viewer than they would be when no bias is applied. A device supports Z biasing if the DDDP::RasterCapsZBias bit of D3DCAPS9::RasterCaps is set.

#define D3DPRASTERCAPS_ZBIAS 0x00004000L

For situations where a polygon offset is needed but the device does not support Z bias, you can achieve the same affect by adjusting the near and far planes for your projection matrix.

**Filling and Reading the Z Buffer**

There are times when you want to fill the Z buffer or read back its contents. However, unless you use a lockable Z buffer format, this cannot be done directly. You can fill the Z buffer by rendering primitives to obtain the desired values in the Z buffer. You can render the primitives with writes to the color buffer disabled so that only the Z buffer is affected by the rendering. The primitives could be a points, lines, or a triangular mesh, depending on the accuracy needed. Points are the most accurate since you can place a point exactly on each pixel in the render target with a specific depth, but they are also the most expensive for large render targets.

Reading back the contents of the Z buffer without actually locking the Z buffer is more complicated. The idea is to render primitives that are affected by the contents of the Z buffer without changing it. You can render planes of different colors parallel to the viewer that completely cover the render target. Setting the depth test to reject pixels that are farther from the viewer than the stored depth value will result in the plane’s color covering the portion of the frame buffer that is farther from the viewer than the depth of the plane. By drawing a sequence of planes from back to front this will populate the render target’s color buffer with colors that correspond to the depth stored in the depth buffer. You can than read the color buffer and obtain an approximation of the contents of the depth buffer.

Both of these techniques can be expensive when high resolution of the depth buffer is needed. In general, the depth buffer should be treated as an opaque chunk of memory that can be neither read or written directly. Hardware manufacturers are continuing to improve the performance of visibility algorithms by using proprietary formats for the depth buffer that do not map directly to the idea of a pixel surface containing a single depth value. If possible, you should use alternative algorithms that do not require explicit reading or writing of the depth buffer.

**14.5 Stencil Test**

The stencil buffer is used in conjunction with the Z buffer. The stencil buffer can provide arbitrary rejection of source pixels based on the results of the depth test combined with the result of the stencil test. The stencil test compares
the value of the current stencil reference value with the stencil buffer value at the destination pixel location. Both stencil values are bitwise ANDed with the current stencil mask so that only specific bits of the stencil values are used in the comparison.

The stencil buffer is part of the Z buffer and is present when using one of the formats D3DFMT_Z24S8, D3DFMT_Z15S1, or D3DFMT_Z24X4S4 for the Z buffer surface format. Otherwise, there is no stencil buffer present and all stencil operations are ignored.

The stencil buffer test is enabled or disabled with RS Stencil Enable. When enabled, RS Stencil Mask controls which bits of the stencil reference value and the destination stencil buffer value are used in the comparison. Meaningful values for RS Stencil Mask are drawn from the interval \([0, 2^s - 1]\), where \(s\) is the depth of the stencil buffer. Any bits in the stencil mask outside this range are ignored. RS Stencil Func defines the comparison function used to evaluate the test and is a member of the enumerated type D3DCMPFUNC. If the result of the comparison function evaluates to FALSE, the source pixel is discarded and no further processing occurs.

New values computed for the stencil buffer itself are computed according to one of three possible cases for a source pixel:

- The stencil test failed.
- The stencil test passed and the depth test failed.
- The stencil test passed and the depth test passed.

The render states RS Stencil Fail, RS Stencil Z Fail, and RS Stencil Pass, respectively, define the operation performed on the existing stencil buffer value to compute a source stencil value for each of the three cases. The values for these render states are given by the D3DSTENCILOP enumerated type and their operation is summarized in table 14.5.

```c
typedef enum _D3DSTENCILOP {
    D3DSTENCILOP_KEEP = 1,
    D3DSTENCILOP_ZERO = 2,
    D3DSTENCILOP_REPLACE = 3,
    D3DSTENCILOP_INVERT = 6,
    D3DSTENCILOP_INCR = 7,
    D3DSTENCILOP_DECR = 8,
    D3DSTENCILOP_INCRSAT = 4,
    D3DSTENCILOP_DECRSAT = 5
} D3DSTENCILOP;
```

Each bit of D3DCAPS9::StencilCaps that is set corresponds to a stencil operation supported by the device. The bits are given by the D3DSTENCILOP flags.

1Calling Clear with D3DCLEAR_STENCIL when no stencil buffer is present is an error.
### 14.5. STENCIL TEST

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DSTENCILOP_KEEP</td>
<td>Keeps the existing destination stencil buffer value as the source stencil value.</td>
</tr>
<tr>
<td>D3DSTENCILOP_ZERO</td>
<td>Uses a value of zero as the source stencil value.</td>
</tr>
<tr>
<td>D3DSTENCILOP_REPLACE</td>
<td>Uses the stencil reference value as the source stencil value.</td>
</tr>
<tr>
<td>D3DSTENCILOP_INVERT</td>
<td>Inverts the destination stencil buffer value as the source stencil value. That is, the source value is the bitwise complement of the destination value.</td>
</tr>
<tr>
<td>D3DSTENCILOP_INCR</td>
<td>Increments the destination stencil buffer value and uses that as the source stencil value, wrapping around to zero when the destination value is the maximum value.</td>
</tr>
<tr>
<td>D3DSTENCILOP_DECR</td>
<td>Decrements the destination stencil buffer value and uses that as the source stencil value, wrapping around to the maximum value when the existing value is zero.</td>
</tr>
<tr>
<td>D3DSTENCILOP_INCRSAT</td>
<td>As D3DSTENCILOP_INCR, but clamp to the maximum stencil value $2^s$.</td>
</tr>
<tr>
<td>D3DSTENCILOP_DECRSAT</td>
<td>As D3DSTENCILOP_DECR, but clamp to zero.</td>
</tr>
</tbody>
</table>

Table 14.1: Summary of the stencil operations corresponding to D3DSTENCILOP.
The stencil buffer can be used to render a variety of special effects. The SDK includes several samples that show a variety of effects: the Stencil Depth sample shows how the stencil buffer can be used to visualize the depth complexity of a scene, the Stencil Mirror sample shows how to use the stencil buffer to create a mirror plane, and the Shadow Volume sample shows how to use the stencil buffer for creating a shadow effect. The following subsections, while not an exhaustive list, describe some additional effects that are commonly used with the stencil buffer.

**Masking Irregular Regions**

A common application of the stencil buffer is to mask irregular regions in the render target. First, we create a mask image in the stencil buffer and then we draw with geometry to be masked with the stencil test set to reject any source pixels where the mask is set.

To create the mask, first clear the stencil buffer to zero. Next, set the rendering state so that any source pixels resulting from rendering will store a 1 into the stencil buffer. This will create a mask where geometry is drawn. Since the alpha test rejects pixels before the stencil test is performed, complex shapes can be drawn using a simple textured quadrilateral with transparent regions in the texture.

```
#define D3DSTENCILCAPS_KEEP 0x00000001L
#define D3DSTENCILCAPS_ZERO 0x00000002L
#define D3DSTENCILCAPS_REPLACE 0x00000004L
#define D3DSTENCILCAPS_INVERT 0x00000020L
#define D3DSTENCILCAPS_INCR 0x00000040L
#define D3DSTENCILCAPS_DECR 0x00000080L
#define D3DSTENCILCAPS_INCRSAT 0x00000008L
#define D3DSTENCILCAPS_DECRSAT 0x00000010L
```

Next, set the rendering state so that geometry will be clipped by the mask. Use the stencil test to reject any source pixels where the destination stencil value is 1. Keep the contents of the stencil buffer will be unaltered by rendering to retain the mask. If you only clear the stencil buffer when you change the mask, you can reuse the mask from frame to frame once it has been created in the frame buffer.
14.6. ALPHA BLENDING

RS Stencil Enable = TRUE
RS Stencil Func = D3DCMP_NOTEQUAL
RS Stencil Ref = 0
RS Stencil Fail = D3DSTENCILOP_KEEP
RS Stencil Z Fail = D3DSTENCILOP_KEEP
RS Stencil Pass = D3DSTENCILOP_KEEP

Screen Door Transparency and Stippling
You can give the appearance of looking through a screen door mesh by setting
the stencil buffer to alternating values of 0 and 1 at every pixel and using
the stencil buffer as a mask. The “screen” is drawn on the pixels where the stencil
buffer has a value of 0 and the scene showing through the screen is drawn where
the stencil buffer has a value of 1. Superior rendering quality can be obtained
by using alpha blending for transparency, but the stencil buffer can provide
an alternative effect. This technique can also be used to provide an arbitrary
stippling pattern in screen space for primitives. One value is used to represent
the “on” portions of the stipple pattern and the other value is used to represent
the “off” portions of the stipple pattern.

Filling and Reading the Stencil Buffer
As the stencil buffer is part of the Z buffer, the same problems for reading and
writing the Z buffer apply to reading and writing the stencil buffer. The same
approaches used for the Z buffer can be used to read or write the contents of
the stencil buffer. The Stencil Depth sample in the SDK uses planes of color to
visualize the contents of the stencil buffer.

14.6 Alpha Blending

Alpha blending combines source pixels and destination pixels based on the al-
pha value of the source and destination pixel and a combining function. The
most common uses of alpha blending are to create a layered compositing effect
wheraby new pixels are blended on top of existing pixels in the render target.

RS Alpha Blend Enable controls the application of alpha blending. If alpha
blending is disabled, then no blending is performed and the source pixel’s alpha
and color will be passed on to the render target directly. If enabled, the source
pixel’s color and destination pixel’s color are first modulated by a selectable
factor and then combined through a function of two arguments:

\[
C = \langle r, g, b, a \rangle \\
f = \langle fr, fg, fb, fa \rangle \\
Cf = \langle rf, gf, fb, fa \rangle \\
\]

\[
C' = \langle func \rangle(Cs, fs, Cd, fd) \\
C' = \langle func \rangle(Cs, fs, Cd, fd) \\
\]
where \(C_s\) is the source pixel’s RGBA color, \(f_s\) is the source blending factor, \(C_d\) is the destination pixel’s RGBA color, \(f_d\) is the destination blending factor, \(\langle \text{func} \rangle\) is the blend function and \(C'_d\) is the result of the alpha blending operation that will be used in further frame buffer processing. In this equation, the colors and the blend factors are both 4D vectors. The blend factors can modulate each color channel separately. When the destination pixel comes from a render target that has no alpha channel, an alpha value of 255, or fully opaque, is provided.

\[
\begin{align*}
\langle \text{func} \rangle (C_s, f_s, C_d, f_d) & = C'_d \\
& = \langle \text{func} \rangle (C_s, f_s, C_d, f_d) \\
& = \langle \text{func} \rangle (C_s, f_s, C_d, f_d)
\end{align*}
\]

\[\text{RS Src Blend} \quad \text{and} \quad \text{RS Dest Blend}\] select the source and destination blend factors \(f_s\) and \(f_d\), respectively. Each has a value from the D3DBLEND enumeration. The blend factors corresponding to each enumerant are given in table 14.2. The blend function \(\langle \text{func} \rangle\) is specified by \(\text{RS Blend Op}\) as one of the enumerants of D3DBLENDOP. The default value is D3DBLENDOP_ADD. The functions corresponding to the enumerants are listed in table 14.3.

```c
typedef enum _D3DBLEND {  
  D3DBLEND_ZERO = 1,  
  D3DBLEND_ONE = 2,  
  D3DBLEND_SRCCOLOR = 3,  
  D3DBLEND_INVSRCOLOR = 4,  
  D3DBLEND_SRCCALPHA = 5,  
  D3DBLEND_INVSRCALPHA = 6,  
  D3DBLEND_DESTALPHA = 7,  
  D3DBLEND_INVDESTALPHA = 8,  
  D3DBLEND_DESTCOLOR = 9,  
  D3DBLEND_INVDESTCOLOR = 10,  
  D3DBLEND_SRCALPHASAT = 11,  
  D3DBLEND_BOTHINVSRCALPHA = 13  
} D3DBLEND;
```

```c
typedef enum _D3DBLENDOP {  
  D3DBLENDOP_ADD = 1,  
  D3DBLENDOP_SUBTRACT = 2,  
  D3DBLENDOP_REVSUBTRACT = 3,  
  D3DBLENDOP_MIN = 4,  
  D3DBLENDOP_MAX = 5  
} D3DBLENDOP;
```

Alpha blending can be used to perform the compositing operations shown in table 14.4 by selecting the appropriate source and destination blend factors with the addition blend operator. A commonly used compositing operator is the “over” operator, used to composite layers rendered from back to front. The operators listed in the table assume that the alpha of the color has been pre-multiplied into the color channels.

When operating in exclusive mode with D3DSWAP_EFFECT_FLIP or D3DSWAP_EFFECT_DISCARD, a device will properly support destination alpha if the D3D_CAPS3_ALPHA_FULLSCREEN_FLIP_OR_DISCARD bit of D3DCAPS9::Caps3 is set. Otherwise, an application should use D3DSWAP_EFFECT_COPY or D3DSwapEffectCopyVSync.

TODO: FIX VSYNC
Enumerant Blend Factor $f$

<table>
<thead>
<tr>
<th>Enumerant</th>
<th>Blend Factor $f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DBLEND_ZERO</td>
<td>$(0, 0, 0, 0)$</td>
</tr>
<tr>
<td>D3DBLEND_ONE</td>
<td>$(1, 1, 1, 1)$</td>
</tr>
<tr>
<td>D3DBLEND_SRCCOLOR</td>
<td>$(r_s, g_s, b_s, a_s)$</td>
</tr>
<tr>
<td>D3DBLEND_INSVRCCOLOR</td>
<td>$(1 - r_s, 1 - g_s, 1 - b_s, 1 - a_s)$</td>
</tr>
<tr>
<td>D3DBLEND_SRCCALPHA</td>
<td>$(a_s, a_s, a_s, a_s)$</td>
</tr>
<tr>
<td>D3DBLEND_INSVRSCALPHA</td>
<td>$(1 - a_s, 1 - a_s, 1 - a_s, 1 - a_s)$</td>
</tr>
<tr>
<td>D3DBLEND_DESTALPHA</td>
<td>$(a_d, a_d, a_d, a_d)$</td>
</tr>
<tr>
<td>D3DBLEND_INVDESTALPHA</td>
<td>$(1 - a_d, 1 - a_d, 1 - a_d, 1 - a_d)$</td>
</tr>
<tr>
<td>D3DBLEND_DESTCOLOR</td>
<td>$(r_d, g_d, b_d, a_d)$</td>
</tr>
<tr>
<td>D3DBLEND_INVDESTCOLOR</td>
<td>$(1 - r_d, 1 - g_d, 1 - b_d, 1 - a_d)$</td>
</tr>
<tr>
<td>D3DBLEND_SRCCALPHASAT</td>
<td>$(f, f, f, f)$, $f = \min(a_s, 1 - a_d)$</td>
</tr>
<tr>
<td>D3DBLEND_BOTHINVSRCALPHA</td>
<td>$(1 - a_s, 1 - a_s, 1 - a_s, 1 - a_s)$</td>
</tr>
</tbody>
</table>

Table 14.2: Blend factors corresponding to the enumerants of D3DBLEND. The source pixel color is $(r_s, g_s, b_s, a_s)$ and the destination pixel color is $(r_d, g_d, b_d, a_d)$.

\[ \text{D3DBLEND\_BOTHINVSRCALPHA} \] is only valid for RS Src Blend and overrides any blend factor specified by RS Dest Blend; the blend factors shown are used for the source and destination blend factors, respectively.

<table>
<thead>
<tr>
<th>Enumerant</th>
<th>Blend Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3DBLENDOP_ADD</td>
<td>$(\text{func})(s, d) = s + d$</td>
</tr>
<tr>
<td>D3DBLENDOP_SUBTRACT</td>
<td>$(\text{func})(s, d) = s - d$</td>
</tr>
<tr>
<td>D3DBLENDOP_REVSUBTRACT</td>
<td>$(\text{func})(s, d) = d - s$</td>
</tr>
<tr>
<td>D3DBLENDOP_MIN</td>
<td>$(\text{func})(s, d) = \min(s, d)$</td>
</tr>
<tr>
<td>D3DBLENDOP_MAX</td>
<td>$(\text{func})(s, d) = \max(s, d)$</td>
</tr>
</tbody>
</table>

Table 14.3: Blending functions corresponding to the enumerants of D3DBLEND-OP. The $s$ and $d$ function arguments are RGBA colors treated as 4D vectors. The min and max functions return the minimum and maximum, respectively, of each color component.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Diagram</th>
<th>Operation</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>![Diagram S]</td>
<td>$D$</td>
<td>![Diagram D]</td>
</tr>
<tr>
<td>$f_s$ = 1, $f_d$ = 0</td>
<td>![Diagram S]</td>
<td>$f_s$ = 0, $f_d$ = 1</td>
<td>![Diagram D]</td>
</tr>
<tr>
<td>clear</td>
<td>![Diagram Clear]</td>
<td>$S$ xor $D$</td>
<td>![Diagram S xor D]</td>
</tr>
<tr>
<td>$f_s$ = 0, $f_d$ = 0</td>
<td>![Diagram Clear]</td>
<td>$f_s$ = $1 - \alpha_d$, $f_d$ = $1 - \alpha_s$</td>
<td>![Diagram S xor D]</td>
</tr>
<tr>
<td>$S$ over $D$</td>
<td>![Diagram S over D]</td>
<td>$D$ over $S$</td>
<td>![Diagram D over S]</td>
</tr>
<tr>
<td>$f_s$ = 1, $f_d$ = $1 - \alpha_s$</td>
<td>![Diagram S over D]</td>
<td>$f_s$ = $1 - \alpha_d$, $f_d$ = 1</td>
<td>![Diagram D over S]</td>
</tr>
<tr>
<td>$S$ in $D$</td>
<td>![Diagram S in D]</td>
<td>$D$ in $S$</td>
<td>![Diagram D in S]</td>
</tr>
<tr>
<td>$f_s$ = $\alpha_d$, $f_d$ = 0</td>
<td>![Diagram S in D]</td>
<td>$f_s$ = 0, $f_d$ = $\alpha_s$</td>
<td>![Diagram D in S]</td>
</tr>
<tr>
<td>$S$ out $D$</td>
<td>![Diagram S out D]</td>
<td>$D$ out $S$</td>
<td>![Diagram D out S]</td>
</tr>
<tr>
<td>$f_s$ = $1 - \alpha_d$, $f_d$ = 0</td>
<td>![Diagram S out D]</td>
<td>$f_s$ = 0, $f_d$ = $1 - \alpha_s$</td>
<td>![Diagram D out S]</td>
</tr>
<tr>
<td>$S$ atop $D$</td>
<td>![Diagram S atop D]</td>
<td>$D$ atop $S$</td>
<td>![Diagram D atop S]</td>
</tr>
<tr>
<td>$f_s$ = $\alpha_d$, $f_d$ = $1 - \alpha_s$</td>
<td>![Diagram S atop D]</td>
<td>$f_s$ = $1 - \alpha_d$, $f_d$ = $\alpha_s$</td>
<td>![Diagram D atop S]</td>
</tr>
</tbody>
</table>

Table 14.4: Compositing operations and their corresponding source and destination blend factors, assuming pre-multiplied alpha values.
14.7. Dithering

All pixel processing conceptually occurs with RGBA colors with at least 8 bits per channel. The render target can have considerably fewer bits per channel, such as D3DFMT_R3G2B2. Just before the color is to be written into the render target, Direct3D will reduce the depth of the color channels to match the render target. This can introduce considerable banding artifacts in smooth color gradients when the render target has a reduced color channel depth. Dithering algorithms can reduce the perceived banding considerably.

Dithering operates by computing the error between the original color and the color to be written into the render target. As pixels are written into the render target, the dithering algorithm attempts to distribute this error evenly across the render target surface. There are a large variety of dithering algorithms in the computer graphics literature. Graphics hardware typically uses a variation of the ordered dither algorithm because of its simplicity in implementation and predictability of dithering artifacts.

Dithering is enabled by RS Dither Enable. If the D3DPRASTERCAPS_DITHER bit of D3DCAPS9::RasterCaps is set, the device supports dithering.

#define D3DPRASTERCAPS_DITHER 0x00000001L
14.8 Multisampling

With multisampling, multiple color, depth and stencil buffer samples are created for each pixel in the render target. During presentation, the multiple samples are combined to provide full-scene antialiasing and other special effects such as depth-of-field, soft shadows and motion blur. When multisampling is used, the device must be using the discard swap effect in the presentation parameters passed to `CreateDevice` or `Reset`. Direct3D supports from 2 to 16 samples per render target pixel and a device advertises its support for multisampling through the `CheckDeviceMultiSampleType` method of the `IDirect3D9` interface.

Full-scene antialiasing is the most common use of multisampling. `RS Multi Sample Antialias` enables full-scene antialiasing when set to `TRUE` by writing to all samples of a multisample render target. The resulting samples are combined in a weighted average to produce smoother polygon edges and silhouettes. Note that a single color value is used for all samples, it is the coverage of the subsamples that is changed through multisampling. Thus, pixel shaders and fixed-function texture processing are executed once for each pixel, not once for each sample in the pixel. However, depth and stencil information is processed for each sample.

When `RS Multi Sample Antialias` is `FALSE`, individual samples in the multisample render target may be controlled through `RS Multi Sample Mask`. Each sample corresponds to a bit in `RS Multi Sample Mask`, starting with the least significant bit. Special effects can be achieved by performing multiple rendering passes and enabling different samples for each pass.

If there are `n` samples in a multisample render target, and `i` samples are enabled during rendering, then the resulting image will have an intensity of `i/n` after presentation. If all samples are enabled, the image will have the same intensity as if no multisampling were used.

If the `DDDPRasterCapsStretchBltMultiSample` bit of `D3DCAPS9::RasterCaps` is set, then the device performs multisampling through a `StretchBlt`-like operation during presentation. In this form of multisampling, the device does not support the masking of individual samples, `RS Multi Sample Mask` has no effect, and the device cannot toggle the state of `RS Multi Sample Antialias` during the rendering of a scene. This implies that full-scene antialiasing can only be toggled on a per-frame basis and not on a per-primitive basis when this bit is set.

```c
#define D3DPRASTERCAPS_STRETCHBLTMULTISAMPLE 0x00800000L
```

Depth of Field

A real camera has a range of depth within the scene where objects are in focus. The range is called the depth of field. The camera model of Direct3D acts as if it has an infinite depth of field, like a pinhole camera. In this case the entire scene is in focus. You can simulate a finite depth of field effect with multisampling using the multisample mask. The idea is to draw the scene in several passes,
14.8. **MULTISAMPLING**

jittering the view frustum slightly during each pass. The results of each pass are combined through multisampling during presentation.

Suppose we have a render target with 16 samples. We can render the scene with 16 passes, enabling a different sample with RS *Multi Sample Mask* during each pass. Each pass uses a slightly different view frustum, but each view frustum intersects the same rectangular region at the same depth from the viewer. The rectangular region in common between all the view frustums is the focal plane where we want all objects to be in focus. Objects distant from the focal plane will be smeared by the jittered view frustums, resulting in an out-of-focus appearance after presentation.

For this effect, it is desirable to have all passes equally weighted, regardless of how many samples are present. To achieve an equal weighting, the same number, or approximately the same number, of samples should be enabled for each pass. Fewer than 16 passes could be used with a fewer number of samples, or groups of samples could be enabled for each pass with 16 samples and a smaller number of passes.

**Motion Blur**

When objects in the scene are moving at a very fast rate compared to the presentation rate, their motion will exhibit temporal aliasing. A common example is the wheels of wagons shown in movies depicting the American southwest in the 1800s. The spokes of the wheels are moving very rapidly compared to the shutter speed, with the shutter acting as a sampling mechanism that samples the scene over time. In computer graphics, the act of sampling the scene over time occurs through rendering. Each frame in a dynamic rendering captures a single instant in time, but does not account for the motion of objects within the scene during the interval between the previously rendered instant and the current instant.

Very fast moving objects that transition linearly through the scene may appear to simply flash once or twice as they speed in front of the virtual camera. A real camera keeps the shutter open for a fixed duration of time and the image of the moving object results in a blurred streak being imaged on the film of the camera. We can simulate this effect with multisampling by rendering very fast moving objects multiple times per frame, storing each rendering into a subset of the samples.

With depth of field effects, we enabled the same number of samples for each rendering pass, giving an equal weighting to each pass. For motion blur, we can do the same and give equal weighting to each rendering pass resulting in a smooth blur as the object moves during the time interval represented by the frame. However, we might also want to weight the rendering passes representing the “older” images of the object lower and weight the “newer” passes higher.

We want to divide the duration of time $t$ represented by the frame into a number of subintervals $N$, each representing the duration $t/N$. With a total of $S$ samples, enabling $S/N$ samples during each rendering pass will give an equal weighting to each pass, as we did for depth of field. To weight “newer” passes...
higher in the final rendering, we enable more samples and fewer samples for the older passes. For instance, suppose there were 16 samples and we were going to motion blur a fast moving object with 4 rendering passes. We could enable 1, 2, 4, and 8 samples for each pass, drawing the passes in the order of increasing time. This would give the heaviest weighting (a full one half of the available samples) to the rendering pass considered most recent and the least weighting to the rendering pass considered the least recent.

14.9 Writing to the Render Target

After all frame buffer processing has been applied to a source pixel, its associated depth, stencil, and color data can be written into the render target. The data written is controlled through a collection of write masks.

RS Z Write Enable determines if depth and stencil values are written into the depth/stencil buffer associated with the render target. The depth test itself is enabled independently of the writing of new depth values, allowing the test to reject pixels even when the depth buffer itself remains unmodified. When depth/stencil buffer writes are enabled, RS Stencil Write Mask controls which bits of the stencil buffer are modified by the stencil operation.

Individual color channels of the render target can be selected for writing with RS Color Write Enable. This render state is a mask containing one or more \texttt{D3DCOLORWRITEENABLE} flag bits. Each set bit corresponds to a channel that is enabled for writing.

\begin{verbatim}
#define D3DCOLORWRITEENABLE_RED   (1L<<0)
#define D3DCOLORWRITEENABLE_GREEN (1L<<1)
#define D3DCOLORWRITEENABLE_BLUE  (1L<<2)
#define D3DCOLORWRITEENABLE_ALPHA (1L<<3)
\end{verbatim}

The color write mask is supported if the \texttt{D3DPMISCCAPS\_COLORWRITEENABLE} bit of \texttt{D3DCAPS9\_PrimitiveMiscCaps} is set.

\begin{verbatim}
#define D3DPMISCCAPS\_COLORWRITEENABLE 0x00000080L
\end{verbatim}

Using alpha blending with RS Src Blend and RS Dest Blend set to \texttt{D3DBLEND\_ZERO} and \texttt{D3DBLEND\_ONE}, respectively, will result in no change to the color buffer of the render target. Most drivers recognize this blend combination and avoid any work in performing the blending, allowing applications to modify the depth and stencil buffers even when support for RS Color Write Enable is not present.

RS Multi Sample Mask controls which samples of a multisample render target are written. The $i$th bit of the mask, where the least significant bit is numbered 0, enables writing to the $i$th sample in the render target.

14.10 \texttt{rt\_FrameBuffer} Sample Application

This sample application demonstrates frame buffer processing by drawing a teapot over a background. The sample demonstrates the alpha test, the depth
test, the stencil test, stencil masking, stencil stippling, alpha blending, dithering, multisample antialiasing, multisample motion blur effects, multisample depth of field effects and frame buffer write masks.

The entire source code is in the code accompanying this book. Listed here is `rt_FrameBuffer.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. The source code contains their definitions.

The alpha test is used in the `render_stencil_stipple` routine on lines 133–172 to reject transparent pixels so that the only portions of the stencil buffer that are written correspond to the opaque areas of the stipple texture. The depth test is configured in the usual manner for visibility testing with no bias in lines 457–459.

The use of stencil planes is demonstrated in several places. Setting the state to clip geometry against nonzero stencil plane values is shown in lines 406–428. Writing values into the stencil planes using geometry is demonstrated in the routines `render_stipple_mask` on lines 98–126 and `render_stencil_stipple` on lines 133–172. The former renders a torus to create an irregular mask consisting of a disc. The latter renders a stipple pattern over the entire back buffer.

Alpha blending is demonstrated through the use of a variety of compositing operators that can be applied to the teapot and the background. By changing the diffuse colors and background colors to an alpha value less than fully opaque, the effect of different blending modes can be visualized. Blending modes requiring destination alpha are disabled when no destination alpha is present in the back buffer.

Dithering, color write enables and Z buffer write enables can be toggled. The effects of dithering can be most readily seen in a 16 bit color display mode. Disable dithering and move the teapot with the arrow keys to see the banding artifacts. Enabling dithering will reduce the banding artifacts considerably.

Multisampling techniques for antialiasing, depth of field effects, and motion blur effects are demonstrated. Antialiasing is shown by setting the appropriate render state on lines 333–335. Motion blur is demonstrated in the routine `render_motion_blur` on lines 246–266. The approach used keeps the x and y axis rotation angle for the previous frame and the current frame, interpolating between the two for each rendering pass. Depth of field is demonstrated in the routine `render_depth_of_field` on lines 274–307. Here, the viewpoint used to define the viewing matrix is jittered off the z axis as the scene is rendered multiple times. The rendered scene consists of 9 teapots instead of a single teapot to more readily visualize the effect.

Back buffers with destination alpha and depth buffers with stencil planes may not be supported on your graphics hardware, particularly if you have an older graphics card. In these situations the program will select the reference rasterizer to provide these features. However, the program also accepts two command line switches that eliminate the requirement of these features. The switches are summarized in table 14.5. The corresponding menu items will be
CHAPTER 14. THE FRAME BUFFER

-`nodest` Do not require destination alpha.
-`nostencil` Do not require stencil planes.

Table 14.5: Command-line switches supported by `rt_FrameBuffer`.

disabled if these command-line switches are used.

Listing 14.1: `rt_FrameBuffer.cpp`: Demonstration of frame buffer processing: alpha blending, stencil masking, stencil stippling, multisample antialiasing, multisample depth of field effects, multisample motion blur effects, dithering and frame buffer write masks.

```cpp
1  //--------------------------------------------------------------------------
2  // File: rt_FrameBuffer.cpp
3  //
4  // Desc: DirectX window application created by the DirectX AppWizard
5  //--------------------------------------------------------------------------
6  #include <algorithm>
7  #include <cmath>
8  #include <sstream>
9  #include <vector>
10 #define STRICT
11 #include <windows.h>
12 #include <windowsx.h>
13 #include <commctrl.h>
14 #include <atlbase.h>
15 #include <d3dx9.h>
16 #include "DXUtil.h"
17 #include "D3DEnumeration.h"
18 #include "D3DSettings.h"
19 #include "D3DApp.h"
20 #include "D3DFont.h"
21 #include "D3DUtil.h"
22 #include "rt/app.h"
23 #include "rt/hr.h"
24 #include "rt/mat.h"
25 #include "rt/mesh.h"
26 #include "rt/misc.h"
27 #include "rt/rt_ColorSel.h"
28 #include "rt/states.h"
29 #include "rt/surface.h"
```
#include "rt/framebuffer.h"

#include "resource.h"

#include "rt_FrameBuffer.h"

const UINT CMyD3DApplication::STIPPLE_SIZE = 64;

const UINT CMyD3DApplication::TILE_SIZE = 64;

/////////////////////////////////////////////////////////////////////
// CMyD3DApplication::s_screen_vertex::FVF
/////////////////////////////////////////////////////////////////////
const DWORD CMyD3DApplication::s_screen_vertex::FVF =
    D3DFVF_XYZRHW | D3DFVF_TEX1 | D3DFVF_TEXCOORDSIZE2(0);

/////////////////////////////////////////////////////////////////////
// g_jitterN, sm_jitter
/////////////////////////////////////////////////////////////////////
const float g_jitter1[2] =
    { 0, 0 };

const float g_jitter2[4] =
    { 0.25f, 0.75f,
      0.75f, 0.25f }

const float g_jitter3[6] =
    { 0.5033922635f, 0.8317967229f,
      0.7806016275f, 0.2504380877f,
      0.2261828938f, 0.4131553612f }

const float g_jitter4[8] =
    { 0.375f, 0.25f,
      0.125f, 0.75f,
      0.875f, 0.25f,
      0.625f, 0.75f }
const float g_jitter5[10] =
{
0.5f, 0.5f,
0.3f, 0.1f,
0.7f, 0.9f,
0.9f, 0.3f,
0.1f, 0.7f
};

const float g_jitter6[12] =
{
0.4646464646f, 0.4646464646f,
0.1313131313f, 0.7979797979f,
0.5353535353f, 0.8686868686f,
0.8686868686f, 0.5353535353f,
0.7979797979f, 0.1313131313f,
0.2020202020f, 0.2020202020f
};

const float g_jitter8[16] =
{
0.5625f, 0.4375f,
0.0625f, 0.9375f,
0.3125f, 0.6875f,
0.6875f, 0.8125f,
0.8125f, 0.1875f,
0.9375f, 0.5625f,
0.4375f, 0.0625f,
0.1875f, 0.3125f
};

const float g_jitter9[18] =
{
0.5f, 0.5f,
0.1666666666f, 0.9444444444f,
0.5f, 0.1666666666f,
0.5f, 0.8333333333f,
0.1666666666f, 0.2777777777f,
0.8333333333f, 0.3888888888f,
0.1666666666f, 0.6111111111f,
0.8333333333f, 0.7222222222f,
0.8333333333f, 0.0555555555f
};

const float g_jitter12[24] =
{
0.4166666666f, 0.625f,
0.9166666666f, 0.875f,
0.25f, 0.375f,
14.10. RT_FRAMEBUFFER SAMPLE APPLICATION

0.4166666666f, 0.125f,
0.75f, 0.125f,
0.0833333333f, 0.125f,
0.75f, 0.625f,
0.25f, 0.875f,
0.5833333333f, 0.375f,
0.9166666666f, 0.375f,
0.0833333333f, 0.625f,
0.5833333333f, 0.875f;

const float g_jitter16[32] =
{
0.375f, 0.4375f,
0.625f, 0.0625f,
0.875f, 0.1875f,
0.125f, 0.0625f,
0.375f, 0.6875f,
0.875f, 0.4375f,
0.625f, 0.5625f,
0.375f, 0.9375f,
0.625f, 0.3125f,
0.125f, 0.5625f,
0.125f, 0.8125f,
0.375f, 0.1875f,
0.875f, 0.9375f,
0.875f, 0.6875f,
0.125f, 0.3125f,
0.625f, 0.8125f
};

const float *
CMyD3DApplication::sm_jitter[16] =
{
g_jitter1,
g_jitter2,
g_jitter3,
g_jitter4,
g_jitter5,
g_jitter6,
g_jitter8,
g_jitter8,
g_jitter9,
g_jitter12,
g_jitter12,
g_jitter12,
g_jitter12,
g_jitter16,
g_jitter16,
g_jitter16,
g_jitter16

};

/////////////////////////////////////////////////////////////////////////
// determine blend factors based on blending operator
const struct CMyD3DApplication::s_blend_factor
CMyD3DApplication::sm_blend_factors[12] =
{
    // in order of e_blending_operator:
    // E_COMPOSITE_CLEAR
    D3DBLEND_ZERO, D3DBLEND_ZERO,
    // E_COMPOSITE_SRC
    D3DBLEND_ONE, D3DBLEND_ZERO,
    // E_COMPOSITE_DEST
    D3DBLEND_ZERO, D3DBLEND_ONE,
    // E_COMPOSITE_SRC_OVER_DEST
    D3DBLEND_ONE, D3DBLEND_INVSRCALPHA,
    // E_COMPOSITE_DEST_OVER_SRC
    D3DBLEND_INVDESTALPHA, D3DBLEND_ONE,
    // E_COMPOSITE_SRC_IN_DEST
    D3DBLEND_DESTALPHA, D3DBLEND_ZERO,
    // E_COMPOSITE_DEST_IN_SRC
    D3DBLEND_ZERO, D3DBLEND_SRCALPHA,
    // E_COMPOSITE_SRC_ATOP_DEST
    D3DBLEND_DESTALPHA, D3DBLEND_INVSRCALPHA,
    // E_COMPOSITE_DEST_ATOP_SRC
    D3DBLEND_INVDESTALPHA, D3DBLEND_SRCALPHA,
    // E_COMPOSITE_SRC_OUT_DEST
    D3DBLEND_INVDESTALPHA, D3DBLEND_ZERO,
    // E_COMPOSITE_DEST_OUT_SRC
    D3DBLEND_ZERO, D3DBLEND_INVSRCALPHA,
    // E_COMPOSITE_SRC_XOR_DEST
    D3DBLEND_INVDESTALPHA, D3DBLEND_INVSRCALPHA


/////////////////////////////////////////////////////////////////////////
// Global access to the app (needed for the global WndProc())
CMyD3DApplication* g_pApp = NULL;
HINSTANCE g_hInst = NULL;
// parse_command_line
// Parse the command line, looking for the switches:
// -nodest Don’t use destination alpha
// -nostencil Don’t use stencil planes
// Throw up a message box if we encounter something we don’t recognize.
bool parse_command_line(LPCTSTR command_line, DWORD &stencil_bits)
{
    rt::tistringstream args(command_line);
    rt::tstring arg;
    while (!args.eof())
    {
        args >> arg;
        if (_T("-nostencil") == arg)
        {
            stencil_bits = 0;
        }
        else if (arg.length())
        {
            rt::tostringstream barf;
            barf << _T("I did not recognize the switch ") << arg
                << _T(" in the command line
") << command_line
                << _T("Valid switches are:
")
                << _T("-nostencil\tDon’t use stencil planes.\n");
            ::MessageBox(NULL, barf.str().c_str(),
                _T("rt_FrameBuffer: Invalid Command Line Argument"), MB_OK);
            return false;
        }
    }
    return true;
}
DWORD stencil_bits = 2;
if (!parse_command_line(command_line, stencil_bits))
{
    return 0;
}
try
{
    CMyD3DApplication d3dApp(stencil_bits);
    g_pApp = &d3dApp;
    g_hInst = instance;
    ::InitCommonControls();
    if (FAILED(d3dApp.Create(instance)))
    {
        return 0;
    }
    return d3dApp.Run();
} catch (rt::hr_message &bang)
{
    return rt::display_error(bang);
} catch (...)
{
    return E_UNEXPECTED;
}

HMENU
find_menu(HMENU menu, const rt::tstring &text)
{
    UINT count = ::GetMenuItemCount(menu);
    for (UINT i = 0; i < count; i++)
    {
        MENUITEMINFO info = { sizeof(info), MIIM_TYPE | MIIM_SUBMENU };
        TWS(::GetMenuItemInfo(menu, i, TRUE, &info));
        if ((MFT_STRING == info.fType) && info.hSubMenu)
        {
            std::vector<TCHAR> buff(info.cch+2);
            info.dwTypeData = &buff[0];
            info.cch = buff.size();
            TWS(::GetMenuItemInfo(menu, i, TRUE, &info));
            if (text == info.dwTypeData)
            {
                return info.hSubMenu;
            }
void enable_menu(HWND menu, const std::wstring &text, bool enabled)
{
    UINT count = GetMenuItemCount(menu);
    for (UINT i = 0; i < count; i++)
    {
        MENUITEMINFO info = {sizeof(info), MIIM_STATE | MIIM_TYPE | MIIM_SUBMENU};
        TWS(::GetMenuItemInfo(menu, i, TRUE, &info));
        if ((MFT_STRING == info.fType) && info.hSubMenu)
        {
            std::vector<wchar_t> buff(info.cch + 2);
            info.dwTypeData = &buff[0];
            info.cch = buff.size();
            TWS(::GetMenuItemInfo(menu, i, TRUE, &info));
            if (text == info.dwTypeData)
            {
                info.fState &= ~(MFS_ENABLED | MFS_DISABLED);
                info.fState |= enabled ? MFS_ENABLED : MFS_DISABLED;
                info.fMask = MIIM_STATE;
                TWS(::SetMenuItemInfo(menu, i, TRUE, &info));
                return;
            }
        }
    }
}

// should have found it
ATLASSERT(false);

// 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(DWORD stencil_bits) :
    CD3DApplication(,}
m_teapot(),
m_texture(),
m_tile(),
m_device_tile(),
m_torus(),
m_stipple_verts(),
m_stipple(),
m_tile_rects(),
m_tile_offsets(),
m_texture_file(_T("dx5_logo.bmp")),
m_num_stipple_quads(0),
m_text_fg(D3DCOLOR_XRGB(255, 255, 0)),
m_specular(D3DCOLOR_XRGB(255, 255, 255)),
m_bg(D3DCOLOR_XRGB(0, 0, 0)),
m_light_color(D3DCOLOR_XRGB(255, 255, 255)),
m_blend_factor(D3DCOLOR_ARGB(128, 128, 128)),
m_color_composite(E_COMPOSITE_NONE),
m_alpha_composite(E_COMPOSITE_NONE),
m_color_blend_op(D3DBLENDOP_ADD),
m_color_src_blend(D3DBLEND_ONE),
m_color_dest_blend(D3DBLEND_ZERO),
m_alpha_src_blend(D3DBLEND_ONE),
m_alpha_dest_blend(D3DBLEND_ZERO),
m_can_separate_alpha_blend(false),
m_color_allow_dest_alpha(false),
m_color_blend_enable(true),
m_alpha_allow_dest_alpha(false),
m_alpha_blend_enable(false),
m_multisample_motion_blur(false),
m_multisample_depth_of_field(false),
m_multisample_antialias(false),
m_color_write_enable_red(true),
m_color_write_enable_green(true),
m_color_write_enable_blue(true),
m_color_write_enable_alpha(true),
m_z_write_enable(true),
m_textured(false),
m_stencil_stippling(false),
m_stencil_mask(false),
m_show_stats(true),
m_animate_view(false),
m_dithered(false),
m_can_texture(false),
m_destination_alpha(false),
m_tile_background(false),
m_build_stencil(false),
m_can_scissor(false),
m_scissor_enable(false),
m_last_rot_x(0.0f),
m_last_rot_y(0.0f),
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 = _T("rt_FrameBuffer");
    m_d3dEnumeration.AppUsesDepthBuffer = TRUE;
    m_bStartFullscreen = false;
    m_bShowCursorWhenFullscreen = false;

    RECT zero = { 0 };
    m_scissor_rect = zero;
    // Read settings from registry
    ReadSettings();
    m_d3dEnumeration.AppMinStencilBits = stencil_bits;

    // Name: ~CMyD3DApplication()
    // Desc: Application destructor. Paired with CMyD3DApplication()
    // Name: OneTimeSceneInit()
    // Desc: Paired with FinalCleanup().
    // The window has been created and the IDirect3D9 interface has been
CHAPTER 14. THE FRAME BUFFER

//@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()
{
    // Drawing loading status message until app finishes loading
    ::SendMessage(m_hWnd, WM_PAINT, 0, 0);
    m_bLoadingApp = FALSE;
    set_menu_data();
    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))
    {
        // Read the stored window width/height. This is just an example,
        // of how to use ::DXUtil_Read*() functions.
        ::DXUtil_ReadIntRegKey(hkey, _T("Width"), &m_dwCreationWidth, m_dwCreationWidth);
        ::DXUtil_ReadIntRegKey(hkey, _T("Height"), &m_dwCreationHeight, m_dwCreationHeight);
        ::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))  
    {  
        // Write the window width/height. This is just an example,  
        // of how to use ::DXUtil_Write*() functions.
        ::DXUtil_WriteIntRegKey(hkey, _T("Width"), m_rcWindowClient.right);  
        ::DXUtil_WriteIntRegKey(hkey, _T("Height"), m_rcWindowClient.bottom);
        
        ::RegCloseKey(hkey);
    }
    
    // Write the window width/height. This is just an example,  
    // of how to use ::DXUtil_Write*() functions.
    
    if (m_can_texture)  
    {  
        create_texture();
        
        DWORD passes = 0;
        const HRESULT hr = m_pd3dDevice->ValidateDevice(&passes);
        if (FAILED(hr))  
        {  
            m_can_texture = false;
        }
    }
if (!m_can_texture)
{
    m_textured = false;
    m_texture = 0;
}

// Create a teapot mesh using D3DX and optimize it
{
    rt::dx_buffer<DWORD> adj;
    THR(::D3DXCreateTeapot(m_pd3dDevice, &m_teapot, &adj));
    THR(m_teapot->OptimizeInplace(D3DXMESHOPT_ATTRSORT, adj,
        NULL, NULL, NULL));
}

// create a background ARGB tile with three steps in alpha in Y
// and a gradient in rgb in X
THR(m_pd3dDevice->CreateOffscreenPlainSurface(TILE_SIZE, TILE_SIZE,
    D3DFMT_A8R8G8B8, D3DPOOL_SCRATCH, &m_tile, NULL));
{
    rt::surface_lock lock(m_tile);
    for (UINT y = 0; y < TILE_SIZE; y++)
    {
        DWORD *row = lock.scanline32(y);
        BYTE alpha = 0;
        if (y < TILE_SIZE/3)
        {
            alpha = 255/3;
        }
        else if (y < 2*TILE_SIZE/3)
        {
            alpha = 255*2/3;
        }
        else
        {
            alpha = 255;
        }
        for (UINT x = 0; x < TILE_SIZE; x++)
        {
            const BYTE rgb = BYTE(alpha*x/(TILE_SIZE-1));
            row[x] = D3DCOLOR_ARGB(alpha, rgb, rgb, rgb);
        }
    }
}
14.10. **RT_FRAMEBUFFER SAMPLE APPLICATION**

587 // create a torus used to create the stencil mask
588 THR(::D3DXCreateTorus(m_pd3dDevice, 0.325f, 0.65f, 10, 50, &m_torus, NULL));

591 // create a stipple texture pattern
592 THR(::D3DXCreateTexture(m_pd3dDevice, STIPPLE_SIZE, STIPPLE_SIZE, 1, 0,
593 D3DFMT_A8R8G8B8, D3DPOOL_MANAGED, &m_stipple));
594 {
595 rt::texture_lock lock(m_stipple);
596 for (UINT y = 0; y < STIPPLE_SIZE; y++)
597 {
598 DWORD *texels = lock.scanline32(y);
599 for (UINT x = 0; x < STIPPLE_SIZE; x++)
600 {
601 if ((y/4) & 1)
602 {
603 texels[x] = ((x/4) & 1) ? 0 : ~0;
604 }
605 else
606 {
607 texels[x] = ((x/4) & 1) ? 0 : ~0;
608 }
609 }
610 }
611 }
612
613 // Init the font
614 THR(m_font.InitDeviceObjects(m_pd3dDevice));
615
616 return S_OK;
617 }
618
619
620
621 //----------------------------------------------------------------------
622 // Name: RestoreDeviceObjects()
623 // Desc: Paired with InvalidateDeviceObjects()
624 // The device exists, but may have just been Reset(). Resources in
625 // D3DPOOL_DEFAULT and any other device state that persists during
626 // rendering should be set here. Render states, matrices, textures,
627 // etc., that don’t change during rendering can be set once here to
628 // avoid redundant state setting during Render() or FrameMove().
629 //----------------------------------------------------------------------
630 HRESULT CMyD3DApplication::RestoreDeviceObjects()
631 {
632
CHAPTER 14. THE FRAME BUFFER

```
HMENU menu = TWS(::GetMenu(m_hWnd));
m_can_scissor = (m_d3dCaps.RasterCaps & D3DPRASTERCAPS_SCISSORTEST) != 0;
rt::enable_menu(menu, ID_OPTIONS_SCISSORTEST, m_can_scissor);
if (m_scissor_enable)
{
    m_scissor_enable = false;
    rt::check_menu(menu, ID_OPTIONS_SCISSORTEST, false);
}
update_blending(menu, false);
m_destination_alpha = (D3DFMT_A8R8G8B8 == m_d3dsdBackBuffer.Format) ||
(D3DFMT_A1R5G5B5 == m_d3dsdBackBuffer.Format) ||
(D3DFMT_A2R10G10B10 == m_d3dsdBackBuffer.Format);
m_can_separate_alpha_blend =
0 != (m_d3dCaps.PrimitiveMiscCaps & D3DPMISCCAPS_SEPARATEALPHABLEND);
update_blending(menu, true);

// if D3DRS_COLORWRITEENABLE is not supported, disable menu items
{
    const bool mask_colors = 0 !=
    (m_d3dCaps.PrimitiveMiscCaps & D3DPMISCCAPS_COLORWRITEENABLE);
    rt::enable_menu(menu, IDM_COLOR_WRITE_ENABLE_RED, mask_colors);
    rt::enable_menu(menu, IDM_COLOR_WRITE_ENABLE_GREEN, mask_colors);
    rt::enable_menu(menu, IDM_COLOR_WRITE_ENABLE_BLUE, mask_colors);
    rt::enable_menu(menu, IDM_COLOR_WRITE_ENABLE_ALPHA, mask_colors);
}

// can't do any stencil effects without stencil bits
{
    const bool support = (m_d3dEnumeration.AppMinStencilBits >= 2);
    if (!support)
    {
        m_stencil_mask = false;
        m_stencil_stippling = false;
        rt::check_menu(menu, IDM_STENCIL_IRREGULAR_MASK, false);
        rt::check_menu(menu, IDM_STENCIL_STIPPLING, false);
    }
    enable_menu(menu, _T("&Stenciling"), support);
}

// can't do separate alpha channel blend
enable_menu(menu, _T("&Alpha Blend"), m_can_separate_alpha_blend);

// can't do any of these without multisampling
if (D3DMULTISAMPLE_NONE == m_d3dpp.MultiSampleType)
```
m_multisample_antialias = false;
m_multisample_depth_of_field = false;
m_multisample_motion_blur = false;

rt::check_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, false);
rt::check_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, false);
rt::check_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, false);

enable_menu(menu, _T("&Multisampling"), false);
//rt::enable_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, false);
//rt::enable_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, false);
//rt::enable_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, false);

}  
#if 0
   // if this cap is set, can't do motion blur or depth of field
else if (m_d3dCaps.RasterCaps & D3DPRASTERCAPS_STRETCHBLTMULTISAMPLE) {
   
m_multisample_depth_of_field = false;
m_multisample_motion_blur = false;

   rt::check_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, false);
   rt::check_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, false);

   rt::enable_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, true);
   rt::enable_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, false);
   rt::enable_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, false);

}  
#endif

else {
   enable_menu(menu, _T("&Multisampling"), true);
   //rt::enable_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, true);
   //rt::enable_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, true);
   //rt::enable_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, true);
}

TWS(::DrawMenuBar(m_hWnd));

// 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.

THR(m_pd3dDevice->SetTransform(D3DTS_VIEW,
   rt::anon(rt::mat_look_at(D3DXVECTOR3(0, 0, -5))));
// Set the projection matrix
D3DXMATRIX matProj;
float aspect = ((float) m_d3dsdBackBuffer.Width)/m_d3dsdBackBuffer.Height;
::D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI/4, aspect, 1.0f, 7.0f);
THR(m_pd3dDevice->SetTransform(D3DTS_PROJECTION, &matProj));

// Set up lighting states
D3DLIGHT9 light;
::D3DUtil_InitLight(light, D3DLIGHT_DIRECTIONAL, -1.0f, -1.0f, 2.0f);
m_pd3dDevice->SetLight(0, &light);
m_pd3dDevice->LightEnable(0, TRUE);
m_pd3dDevice->SetRenderState(D3DRS_LIGHTING, TRUE);

// create a tile in the same format as the back buffer and load it
THR(m_pd3dDevice->CreateOffscreenPlainSurface(TILE_SIZE, TILE_SIZE,
m_d3dsdBackBuffer.Format, D3DPOOL_DEFAULT, &m_device_tile, NULL));
THR(::D3DXLoadSurfaceFromSurface(m_device_tile, NULL, NULL,
m_tile, NULL, NULL, D3DX_FILTER_NONE, 0));

// rebuild stencil planes if needed
m_build_stencil = m_stencil_mask || m_stencil_stippling;

// build RECTs and POINTs for tiling the background
m_tile_rects.clear();
m_tile_offsets.clear();
const UINT width = m_d3dsdBackBuffer.Width;
const UINT height = m_d3dsdBackBuffer.Height;
for (UINT y = 0; y < height; y += TILE_SIZE)
{
    for (UINT x = 0; x < width; x += TILE_SIZE)
    {
        const bool y_even = ((y/TILE_SIZE) & 1) == 0;
        const bool x_even = ((x/TILE_SIZE) & 1) == 0;
        if ((x_even && y_even) || (!x_even && !y_even))
        {
            const POINT offset = { x, y };
            const RECT rect =
            {
                0, 0,
                (x + TILE_SIZE) <= width ? TILE_SIZE : width-x,
                (y + TILE_SIZE) <= height ? TILE_SIZE : height-y
            };
            m_tile_rects.push_back(rect);
            m_tile_offsets.push_back(offset);
// build s_screen_vertex structures for stippling the window
m_stipple_verts = 0;
m_num_stipple_quads = (width + STIPPLE_SIZE - 1)/STIPPLE_SIZE*
( height + STIPPLE_SIZE - 1)/STIPPLE_SIZE;
THR(m_pd3dDevice->CreateVertexBuffer(m_num_stipple_quads*6*sizeof(s_screen_vertex),
D3DUSAGE_WRITEONLY, s_screen_vertex::FVF, D3DPOOL_MANAGED,
&m_stipple_verts, NULL));
rt::vertex_lock<s_screen_vertex> verts(m_stipple_verts);
s_screen_vertex *quad = verts.data();
for (y = 0; y < height; y += STIPPLE_SIZE)
{
    for (UINT x = 0; x < width; x += STIPPLE_SIZE)
    {
        const float right = float(std::min(x + STIPPLE_SIZE, width));
        const float bottom = float(std::min(y + STIPPLE_SIZE, height));
        quad[0].x = x + 0.5f;
        quad[0].y = y + 0.5f;
        quad[0].z = 0.99f;
        quad[0].rhw = 0.99f;
        quad[0].u = 0;
        quad[0].v = 1;
        quad[1].x = right + 0.5f;
        quad[1].y = y + 0.5f;
        quad[1].z = 0.99f;
        quad[1].rhw = 0.99f;
        quad[1].u = 1;
        quad[1].v = 1;
        quad[2].x = x + 0.5f;
        quad[2].y = bottom + 0.5f;
        quad[2].z = 0.99f;
        quad[2].rhw = 0.99f;
        quad[2].u = 0;
        quad[2].v = 0;
        quad[3].x = x + 0.5f;
        quad[3].y = bottom + 0.5f;
        quad[3].z = 0.99f;
        quad[3].rhw = 0.99f;
        quad[3].u = 0;
        quad[3].v = 0;
quad[4].x = right + 0.5f;
quad[4].y = y + 0.5f;
quad[4].z = 0.99f;
quad[4].rhw = 0.99f;
quad[4].u = 1;
quad[4].v = 1;

quad[5].x = right + 0.5f;
quad[5].y = bottom + 0.5f;
quad[5].z = 0.99f;
quad[5].rhw = 0.99f;
quad[5].u = 1;
quad[5].v = 0;
quad += 6;

// Restore the font
THR(m_font.RestoreDeviceObjects());
return S_OK;

// Update the world state according to user input
if (m_animate_view)
{
    m_rot_y += m_fElapsedTime;
    if (m_rot_y > 2*D3DX_PI)
m_rot_y = std::fmodf(m_rot_y, 2*D3DX_PI);
}
else {
    if (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_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;
return S_OK;

void CMyD3DApplication::UpdateInput()
{
    m_input.m_up = (m_bActive && (GetAsyncKeyState(VK_UP) & 0x8000) == 0x8000);
    m_input.m_down = (m_bActive && (GetAsyncKeyState(VK_DOWN) & 0x8000) == 0x8000);
    m_input.m_left = (m_bActive && (GetAsyncKeyState(VK_LEFT) & 0x8000) == 0x8000);
    m_input.m_right = (m_bActive && (GetAsyncKeyState(VK_RIGHT) & 0x8000) == 0x8000);
}
// 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
    THR(m_pd3dDevice->SetRenderState(D3DRS_SCISSORTESTENABLE, FALSE));
    THR(m_pd3dDevice->Clear(0L, NULL, D3DCLEAR_TARGET |
    D3DCLEAR_ZBUFFER, m_bg, 1.0f, 0L));

    if (mTileBackground)
    {
        tile_background();
    }

    // enable or disable full-scene antialiasing
    THR(m_pd3dDevice->SetRenderState(
        D3DRS_MULTISAMPLEANTIALIAS,
        m_multisample_antialias));

    THR(m_pd3dDevice->BeginScene());

    // set up state for rendering: materials & lighting
    D3DMATERIAL9 mtrl;
    ::D3DUtil_InitMaterial(mtrl, 1.0f, 0.0f, 0.0f);
    mtrl.Diffuse = D3DXCOLOR(m_fg);
    mtrl.Specular = D3DXCOLOR(m_specular);
    THR(m_pd3dDevice->SetMaterial(&mtrl));

    // build stencil planes, if necessary
    if (mBuildStencil)
    {
        render_stencil();
        mBuildStencil = false;
    }

    // texturing states
    if (mCanTexture && mTextured)
    {
        const rt::s_tss ts_states[] =
        {
            D3DTSS_TEXCOORDINDEX,
            D3DTSS_TCI_CAMERASPACEPOSITION,
            D3DTSS_COLORARG1, D3DTA_TEXTURE,
D3DTSS_COLOROP, D3DTOP_MODULATE,
D3DTSS_COLORARG2, D3DTA_DIFFUSE,
D3DTSS_ALPHAARG1, D3DTA_TEXTURE,
D3DTSS_ALPHAO, D3DTOP_SELECTARG2,
D3DTSS_ALPHARG2, D3DTA_DIFFUSE,
D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2
};
rt::set_states(m_pd3dDevice, 0, ts_states,
NUM_OF(ts_states));
const rt::s_ss samp_states[] =
{
    D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP,
    D3DSAMP_ADDRESSV, D3DTADDRESS_WRAP,
    D3DSAMP_MINFILTER, D3DTEXF_LINEAR,
    D3DSAMP_MAGFILTER, D3DTEXF_LINEAR,
    D3DSAMP_MIPFILTER, D3DTEXF_LINEAR
};
rt::set_states(m_pd3dDevice, 0, samp_states,
NUM_OF(samp_states));
ATLASSERT(m_texture);
THR(m_pd3dDevice->SetTexture(0, m_texture));
// set a (u,v) transform that makes the flag
// rectangular and positioned nicely on the
// teapot. Invert the y-axis to get it in
// the proper vertical orientation. Rotate
// it around the Z axis to make it a little
// more interesting.
D3DXMATRIX texform =
    rt::mat_rot_z(D3DX_PI/4.0f)*
    rt::mat_scale(0.5f, -1, 1)*
    rt::mat_trans(-0.6f, -0.5f, 0);
THR(m_pd3dDevice->SetTransform(D3DTS_TEXTURE0,
    &texform));
}
else
{
    // disable texturing
    const rt::s_tss ts_states[] =
    {
        D3DTSS_TEXCOORDINDEX, 0,
        D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_DISABLE,
        D3DTSS_COLOROP, D3DTOP_DISABLE,
        D3DTSS_ALPHAO, D3DTOP_DISABLE
    };
    rt::set_states(m_pd3dDevice, 0, ts_states,
    NUM_OF(ts_states));
THR(m_pd3dDevice->SetTexture(0, NULL));
}

// stencil test state
if (m_stencil_mask || m_stencil_stippling)
{
    // draw teapot only where no stencil bits set,
    // and don’t modify the stencil bits
    rt::s_rs states[] =
    {
        D3DRS_STENCILENABLE, true,
        D3DRS_STENCILFUNC, D3DCMP_EQUAL,
        D3DRS_STENCILREF, 0,
        D3DRS_STENCILMASK, ~0UL,
        D3DRS_STENCILWRITEMASK, 0,
        D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP,
        D3DRS_STENCILPASS, D3DSTENCILOP_KEEP,
        D3DRS_STENCILZFAIL, D3DSTENCILOP_KEEP
    };
    rt::set_states(m_pd3dDevice, states,
                   NUM_OF(states));
}
else
{
    THR(m_pd3dDevice->SetRenderState(
       D3DRS_STENCILENABLE, false));
}

// setup blending state
{
    rt::s_rs states[] =
    {
        D3DRS_ALPHABLENDENABLE, m_color_blend_enable,
        D3DRS_SRCCBLEND, m_color_src_blend,
        D3DRS_DESTBLEND, m_color_dest_blend,
        D3DRS_BLENDOP, m_color_blend_op,
        D3DRS_BLENDFACTOR, m_blend_factor,
        D3DRS_SEPARATEALPHABLENDENABLE, m_alpha_blend_enable,
        D3DRS_SRCCBLENDALPHA, m_alpha_src_blend,
        D3DRS_DESTBLENDALPHA, m_alpha_dest_blend,
        D3DRS_BLENDOPALPHA, m_alpha_blend_op
    };
    rt::set_states(m_pd3dDevice, states,
                   NUM_OF(states));
}

// frame buffer render states
const rt::s_rs states[] =
{
    D3DRS_SCISSORTESTENABLE, m_scissor_enable,
    D3DRS_NORMALIZENORMALS, true,
    D3DRS_LIGHTING, true,
    D3DRS_SPECULAREENABLE, true,
    D3DRS_ZENABLE, true,
    D3DRS_ZFUNC, D3DCMP_LESS,
    D3DRS_DITHERENABLE, m_dithered,
    D3DRS_ZWRITEENABLE, m_z_write_enable
};
rt::set_states(m_pd3dDevice, states, NUM_OF(states));

// Render the teapot mesh
if (m_multisample_depth_of_field)
{
    render_depth_of_field();
}
else if (m_multisample_motion_blur)
{
    render_motion_blur();
}
else
{
    render_teapot(rt::mat_rot_x(m_rot_x)*rt::mat_rot_y(m_rot_y));
}

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

THR(m_pd3dDevice->EndScene());
return S_OK;

//////////////////////////////////////////////////////////////////////
// operator<<(e_composite_operator)
//
// Formats an enumerant of e_composite_operator as a string.
//
rt::tostream &
operator<<(rt::tostream &stream, e_composite_operator blending)
{
    switch (blending)
    {
#define BLEND_OUT(enumerant_, str_)
    case enumerant_: stream << _T(str_); break

    BLEND_OUT(E_COMPOSITE_CLEAR, "clear");
    BLEND_OUT(E_COMPOSITE_SRC, "src");
    BLEND_OUT(E_COMPOSITE_DEST, "dest");
    BLEND_OUT(E_COMPOSITE_SRC_OVER_DEST, "src over dest");
    BLEND_OUT(E_COMPOSITE_DEST_OVER_SRC, "dest over src");
    BLEND_OUT(E_COMPOSITE_SRC_IN_DEST, "src in dest");
    BLEND_OUT(E_COMPOSITE_DEST_IN_SRC, "dest in src");
    BLEND_OUT(E_COMPOSITE_SRC_ATOP_DEST, "src atop dest");
    BLEND_OUT(E_COMPOSITE_DEST_ATOP_SRC, "dest atop src");
    BLEND_OUT(E_COMPOSITE_SRC_OUT_DEST, "src out dest");
    BLEND_OUT(E_COMPOSITE_DEST_OUT_SRC, "dest out src");
    BLEND_OUT(E_COMPOSITE_SRC_XOR_DEST, "src xor dest");
    BLEND_OUT(E_COMPOSITE_NONE, "none");

#undef BLEND_OUT
    default:
        ATLASSERT(false);
    }
    return stream;
}"
14.10. RT_FRAMEBUFFER SAMPLE APPLICATION

// Render lines of text at a specific offset and in a specific color.
// Some program options are displayed in addition to the standard
// AppWizard sample text.

void CMYD3DApplication::render_text_aux(float offset_x, float offset_y, D3DCOLOR fg)
{
    // Output display stats
    float y = 2.0f + offset_y;
    float x = 2.0f + offset_x;
    rt::tostringstream msg;

    text(x, y, fg, m_strFrameStats);
    y += 20.0f; text(x, y, fg, m_strDeviceStats);

    msg << (m_destination_alpha ?
            _T("Destination alpha") : _T("No destination alpha"));
    if (m_dithered)
    {
        msg << _T(",", dither");
    }
    if (E_COMPOSITE_NONE != m_color_composite)
    {
        msg << _T("",") << m_color_composite;
    }
    if (m_multisample_antialias)
    {
        msg << _T(",", multisample antialias");
    }
    else if (m_multisample_depth_of_field)
    {
        msg << _T(",", multisample depth of field");
    }
    else if (m_multisample_motion_blur)
    {
        msg << _T(",", multisample motion blur");
    }
    else if (m_stencil_mask)
    {
        msg << _T(",", stencil mask");
    }
    if (m_stencil_stippling)
    {
        msg << _T(",", stencil stipple");
    }
    if (m_scissor_enable)


```cpp
CHAPTER 14. THE FRAME BUFFER

{  
    msg << _T("scissor");
}

msg << std::ends;
y += 20.0f; text(x, y, fg, msg);

if (m_color_blend_enable &&
(D3DBLEND_BLENDFACTOR == m_color_src_blend) ||
(D3DBLEND_BLENDFACTOR == m_color_dest_blend) ||
(D3DBLEND_INVBLENDFACTOR == m_color_src_blend) ||
(D3DBLEND_INVBLENDFACTOR == m_color_dest_blend) ||
(m_alpha_blend_enable &&
(D3DBLEND_BLENDFACTOR == m_alpha_src_blend) ||
(D3DBLEND_BLENDFACTOR == m_alpha_dest_blend) ||
(D3DBLEND_INVBLENDFACTOR == m_alpha_src_blend) ||
(D3DBLEND_INVBLENDFACTOR == m_alpha_dest_blend)))
{
    msg.seekp(0);
    msg << _T("Blend factor: RGBA(")
        << int(rt::D3DColor_Red(m_blend_factor))
        << _T(", ")
        << int(rt::D3DColor_Green(m_blend_factor))
        << _T(", ")
        << int(rt::D3DColor_Blue(m_blend_factor))
        << _T(", ")
        << int(rt::D3DColor_Alpha(m_blend_factor))
        << _T(")")
        << std::ends;
    y += 20.0f; text(x, y, fg, msg);
    // Output statistics & help
    y = float(m_d3dsdBackBuffer.Height + offset_y + 2);
    msg.seekp(0);
    msg << _T("Arrow keys: Up=")
        << int(m_input.m_up)
        << _T(" Down=")
        << int(m_input.m_down)
        << _T(" Left=")
        << int(m_input.m_left)
        << _T(" Right=")
        << int(m_input.m_right)
        << std::ends;
    y -= 20.0f; text(x, y, fg, msg);
    y -= 20.0f; text(x, y, fg, _T("Use arrow keys to rotate object"));
    y -= 20.0f; text(x, y, fg, _T("Press 'F2' to configure display"));
}

// ---\---------------------------------------------------------------------
// Name: MsgProc()
// Desc: Overrides 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,
LRESULT result = 0;
bool handled = false;

switch (msg)
{
    case WM_LBUTTONDOWN:
        result = on_left_button_down(hWnd, wParam, lParam, handled);
        break;

    case WM_MOUSEMOVE:
        result = on_mouse_move(hWnd, wParam, lParam, handled);
        break;

    case WM_LBUTTONUP:
        result = on_left_button_up(hWnd, wParam, lParam, handled);
        break;

    case WM_PAINT:
        if (m_bLoadingApp)
        {
            // Draw on the window tell the user that the app is loading
            HDC hDC = TWS(::GetDC(hWnd));
            RECT rct;
            TWS(::GetClientRect(hWnd, &rct));
            ::DrawText(hDC, _T("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);
}
CHAPTER 14. THE FRAME BUFFER

1277  // Desc: Invalidates device objects. Paired with RestoreDeviceObjects()
1278  //--------------------------------------------------------------------------
1279  HRESULT CMyD3DApplication::InvalidateDeviceObjects()
1280  {
1281      m_device_tile = 0;
1282      m_stipple_verts = 0;
1283      THR(m_font.InvalidateDeviceObjects());
1284      return S_OK;
1285  }
1286
1287  //--------------------------------------------------------------------------
1288  // Name: DeleteDeviceObjects()
1289  // Desc: Paired with InitDeviceObjects()
1290  // Called when the app is exiting, or the device is being changed,
1291  // this function deletes any device dependent objects.
1292  //--------------------------------------------------------------------------
1293  HRESULT CMyD3DApplication::DeleteDeviceObjects()
1294  {
1295      m_teapot = 0;
1296      m_torus = 0;
1297      m_texture = 0;
1298      m_tile = 0;
1299      m_device_tile = 0;
1300      m_stipple = 0;
1301      THR(m_font.DeleteDeviceObjects());
1302      return S_OK;
1303  }
1304
1305  //--------------------------------------------------------------------------
1306  // Name: FinalCleanup()
1307  // Desc: Paired with OneTimeSceneInit()
1308  // Called before the app exits, this function gives the app the chance
1309  // to cleanup after itself.
1310  //--------------------------------------------------------------------------
1311  HRESULT CMyD3DApplication::FinalCleanup()
1312  {
1313      // TODO: Perform any final cleanup needed
1314  }
1323    // Write the settings to the registry
1324    WriteSettings();
1325
1326    return S_OK;
1327 }
1328
1329
1330
1331
1332    //----------------------------------------------------------------------------------------
1333    // CMyD3DApplication::create_texture
1334    //
1335    // Creates the texture map from the given file. If that was successful, it
1336    // becomes the current texture used on the next frame draw.
1337    //
1338    void
1339    CMyD3DApplication::create_texture(LPCTSTR file)
1340    {
1341        CComPtr<IDirect3DTexture9> new_texture;
1342
1343        const HRESULT hr = ::D3DUtil_CreateTexture(m_pd3dDevice,
1344            file ? file : m_texture_file.c_str(),
1345            &new_texture, D3DFMT_A8R8G8B8);
1346
1347        if (SUCCEEDED(hr))
1348            {
1349                m_texture = new_texture;
1350                if (file)
1351                    {
1352                        m_texture_file = file;
1353                    }
1354            }
1355            else
1356                {
1357                    THR(hr);
1358                }
1359    }
1360
1361    template <typename T>
1362    DWORD
1363    find_id(const rt::s_enum_value<T> *items, UINT num_items, T value)
1364    {
1365        for (UINT i = 0; i < num_items; i++)
1366            {
1367                if (value == items[i].m_state)
1368                    return items[i].m_value;
ATLASSERT(false);
return 0;
}

DWORD
color_composite_id(e_composite_operator value)
{
    const rt::s_enum_value<e_composite_operator> items[] =
    {
        E_COMPOSITE_CLEAR, IDM_COLOR_COMPOSITE_CLEAR,
        E_COMPOSITE_SRC, IDM_COLOR_COMPOSITE_SRC,
        E_COMPOSITE_DEST, IDM_COLOR_COMPOSITE_DEST,
        E_COMPOSITE_SRC_OVER_DEST, IDM_COLOR_COMPOSITE_SRC_OVER_DEST,
        E_COMPOSITE_DEST_OVER_SRC, IDM_COLOR_COMPOSITE_DEST_OVER_SRC,
        E_COMPOSITE_SRC_IN_DEST, IDM_COLOR_COMPOSITE_SRC_IN_DEST,
        E_COMPOSITE_DEST_IN_SRC, IDM_COLOR_COMPOSITE_DEST_IN_SRC,
        E_COMPOSITE_SRC_ATOP_DEST, IDM_COLOR_COMPOSITE_SRC_ATOP_DEST,
        E_COMPOSITE_DEST_ATOP_SRC, IDM_COLOR_COMPOSITE_DEST_ATOP_SRC,
        E_COMPOSITE_SRC_OUT_DEST, IDM_COLOR_COMPOSITE_SRC_OUT_DEST,
        E_COMPOSITE_DEST_OUT_SRC, IDM_COLOR_COMPOSITE_DEST_OUT_SRC,
        E_COMPOSITE_SRC_XOR_DEST, IDM_COLOR_COMPOSITE_SRC_XOR_DEST
    };
    return find_id(items, NUM_OF(items), value);
}

DWORD
color_blend_op_id(D3DBLENDOP value)
{
    const rt::s_enum_value<D3DBLENDOP> items[] =
    {
        D3DBLENDOP_ADD, IDM_COLOR_BLEND_OP_ADD,
        D3DBLENDOP_SUBTRACT, IDM_COLOR_BLEND_OP_SUBTRACT,
        D3DBLENDOP_REVSUBTRACT, IDM_COLOR_BLEND_OP_REVERSE_SUBTRACT,
        D3DBLENDOP_MIN, IDM_COLOR_BLEND_OP_MINIMUM,
        D3DBLENDOP_MAX, IDM_COLOR_BLEND_OP_MAXIMUM
    };
    return find_id(items, NUM_OF(items), value);
}

DWORD
color_src_blend_id(D3DBLEND value)
{
    const rt::s_enum_value<D3DBLEND> items[] =
    {
D3DBLEND_ZERO, IDM_COLOR_SRC_BLEND_ZERO,
D3DBLEND_ONE, IDM_COLOR_SRC_BLEND_ONE,
D3DBLEND_SRCCOLOR, IDM_COLOR_SRC_BLEND_SRCCOLOR,
D3DBLEND_INVSRCCOLOR, IDM_COLOR_SRC_BLEND_INVSRCCOLOR,
D3DBLEND_SRCALPHA, IDM_COLOR_SRC_BLEND_SRCALPHA,
D3DBLEND_INVSRCALPHA, IDM_COLOR_SRC_BLEND_INVSRCALPHA,
D3DBLEND_DESTALPHA, IDM_COLOR_SRC_BLEND_DESTALPHA,
D3DBLEND_INVDESTALPHA, IDM_COLOR_SRC_BLEND_INVDESTALPHA,
D3DBLEND_DESTCOLOR, IDM_COLOR_SRC_BLEND_DESTCOLOR,
D3DBLEND_INVDESTCOLOR, IDM_COLOR_SRC_BLEND_INVDESTCOLOR,
D3DBLEND_SRCALPHASAT, IDM_COLOR_SRC_BLEND_SRCALPHASAT,
D3DBLEND_BLENDFACTOR, IDM_COLOR_SRC_BLEND_BLENDFACTOR,
D3DBLEND_INVBLENDFACTOR, IDM_COLOR_SRC_BLEND_INVBLENDFACTOR
};
return find_id(items, NUM_OF(items), value);
}

DWORD
color_dest_blend_id(D3DBLEND value)
{
    const rt::s_enum_value<D3DBLEND> items[] =
    {
        D3DBLEND_ZERO, IDM_COLOR_DEST_BLEND_ZERO,
        D3DBLEND_ONE, IDM_COLOR_DEST_BLEND_ONE,
        D3DBLEND_SRCCOLOR, IDM_COLOR_DEST_BLEND_SRCCOLOR,
        D3DBLEND_INVSRCCOLOR, IDM_COLOR_DEST_BLEND_INVSRCCOLOR,
        D3DBLEND_SRCALPHA, IDM_COLOR_DEST_BLEND_SRCALPHA,
        D3DBLEND_INVSRCALPHA, IDM_COLOR_DEST_BLEND_INVSRCALPHA,
        D3DBLEND_DESTALPHA, IDM_COLOR_DEST_BLEND_DESTALPHA,
        D3DBLEND_INVDESTALPHA, IDM_COLOR_DEST_BLEND_INVDESTALPHA,
        D3DBLEND_DESTCOLOR, IDM_COLOR_DEST_BLEND_DESTCOLOR,
        D3DBLEND_INVDESTCOLOR, IDM_COLOR_DEST_BLEND_INVDESTCOLOR,
        D3DBLEND_SRCALPHASAT, IDM_COLOR_DEST_BLEND_SRCALPHASAT,
        D3DBLEND_BLENDFACTOR, IDM_COLOR_DEST_BLEND_BLENDFACTOR,
        D3DBLEND_INVBLENDFACTOR, IDM_COLOR_DEST_BLEND_INVBLENDFACTOR
    };
    return find_id(items, NUM_OF(items), value);
}

DWORD
alpha_composite_id(e_composite_operator value)
{
    const rt::s_enum_value<e_composite_operator> items[] =
    {
        E_COMPOSITE_CLEAR, IDM_ALPHA_COMPOSITE_CLEAR,
E_COMPOSITE_SRC, IDM_ALPHA_COMPOSITE_SRC,
E_COMPOSITE_DEST, IDM_ALPHA_COMPOSITE_DEST,
E_COMPOSITE_SRC_OVER_DEST, IDM_ALPHA_COMPOSITE_SRC_OVER_DEST,
E_COMPOSITE_DEST_OVER_SRC, IDM_ALPHA_COMPOSITE_DEST_OVER_SRC,
E_COMPOSITE_SRC_IN_DEST, IDM_ALPHA_COMPOSITE_SRC_IN_DEST,
E_COMPOSITE_DEST_IN_SRC, IDM_ALPHA_COMPOSITE_DEST_IN_SRC,
E_COMPOSITE_SRC_ATOP_DEST, IDM_ALPHA_COMPOSITE_SRC_ATOP_DEST,
E_COMPOSITE_DEST_ATOP_SRC, IDM_ALPHA_COMPOSITE_DEST_ATOP_SRC,
E_COMPOSITE_SRC_OUT_DEST, IDM_ALPHA_COMPOSITE_SRC_OUT_DEST,
E_COMPOSITE_DEST_OUT_SRC, IDM_ALPHA_COMPOSITE_DEST_OUT_SRC,
E_COMPOSITE_SRC_XOR_DEST, IDM_ALPHA_COMPOSITE_SRC_XOR_DEST
};
return find_id(items, NUM_OF(items), value);
}

DWORD
alpha_blend_op_id(D3DBLENDOP value)
{
const rt::s_enum_value<D3DBLENDOP> items[] =
{
D3DBLENDOP_ADD, IDM_ALPHA_BLEND_OP_ADD,
D3DBLENDOP_SUBTRACT, IDM_ALPHA_BLEND_OP_SUBTRACT,
D3DBLENDOP_REVSUBTRACT, IDM_ALPHA_BLEND_OP_REVERSE_SUBTRACT,
D3DBLENDOP_MIN, IDM_ALPHA_BLEND_OP_MINIMUM,
D3DBLENDOP_MAX, IDM_ALPHA_BLEND_OP_MAXIMUM
};
return find_id(items, NUM_OF(items), value);
}

DWORD
alpha_src_blend_id(D3DBLEND value)
{
const rt::s_enum_value<D3DBLEND> items[] =
{
D3DBLEND_ZERO, IDM_ALPHA_SRC_BLEND_ZERO,
D3DBLEND_ONE, IDM_ALPHA_SRC_BLEND_ONE,
D3DBLEND_SRCCOLOR, IDM_ALPHA_SRC_BLEND_SRCCOLOR,
D3DBLEND_INVSRCOLOR, IDM_ALPHA_SRC_BLEND_INV_SRCCOLOR,
D3DBLEND_SRCCALPHA, IDM_ALPHA_SRC_BLEND_SRCCALPHA,
D3DBLEND_INVSRCCALPHA, IDM_ALPHA_SRC_BLEND_INV_SRCCALPHA,
D3DBLEND_DESTALPHA, IDM_ALPHA_SRC_BLEND_DESTALPHA,
D3DBLEND_INVDESTALPHA, IDM_ALPHA_SRC_BLEND_INVDESTALPHA,
D3DBLEND_DESTCOLOR, IDM_ALPHA_SRC_BLEND_DESTCOLOR,
D3DBLEND_INVDESTCOLOR, IDM_ALPHA_SRC_BLEND_INVDESTCOLOR,
D3DBLEND_SRCCOLORALPHA, IDM_ALPHA_SRC_BLEND_SRCCOLORALPHA,
D3DBLEND_INVSRCCOLORALPHA, IDM_ALPHA_SRC_BLEND_INVSRCCOLORALPHA,
D3DBLEND_BOTHINVSRCALPHA, IDM_ALPHA_SRC_BLEND_BOTH_INV_SRCCALPHA,
14.10. **RT_FRAMEBUFFER SAMPLE APPLICATION**

1507     D3DBLEND_BLENDFACTOR, IDM_ALPHA_SRC_BLENDFACTOR,
1508     D3DBLEND_INVBLENDFACTOR, IDM_ALPHA_SRC_BLENDFACTOR_INV
1509     );
1510     return find_id(items, NUM_OF(items), value);
1511 }
1512
1513 DWORD
1514 alpha_dest_blend_id(D3DBLEND value)
1515 {
1516     const rt::s_enum_value<D3DBLEND> items[] =
1517     {
1518     D3DBLEND_ZERO, IDM_ALPHA_DEST_BLENDFACTOR_ZERO,
1519     D3DBLEND_ONE, IDM_ALPHA_DEST_BLENDFACTOR_ONE,
1520     D3DBLEND_SRCCOLOR, IDM_ALPHA_DEST_BLENDFACTOR_SRCCOLOR,
1521     D3DBLEND_INVSRCCOLOR, IDM_ALPHA_DEST_BLENDFACTOR_INVSRCCOLOR,
1522     D3DBLEND_SRCALPHA, IDM_ALPHA_DEST_BLENDFACTOR_SRCALPHA,
1523     D3DBLEND_INVSRCALPHA, IDM_ALPHA_DEST_BLENDFACTOR_INVSRCALPHA,
1524     D3DBLEND_DESTALPHA, IDM_ALPHA_DEST_BLENDFACTOR_DESTALPHA,
1525     D3DBLEND_INVDESTALPHA, IDM_ALPHA_DEST_BLENDFACTOR_INV_DESTALPHA,
1526     D3DBLEND_DESTCOLOR, IDM_ALPHA_DEST_BLENDFACTOR_DESTCOLOR,
1527     D3DBLEND_INVDESTCOLOR, IDM_ALPHA_DEST_BLENDFACTOR_INVDESTCOLOR,
1528     D3DBLEND_SRCCOLORALPHA, IDM_ALPHA_DEST_BLENDFACTOR_SRCCOLORALPHA,
1529     D3DBLEND_INVSRCCOLORALPHA, IDM_ALPHA_DEST_BLENDFACTOR_INVSRCCOLORALPHA,
1530     D3DBLEND_BLENDFACTOR, IDM_ALPHA_DEST_BLENDFACTOR_BLENDFACTOR,
1531     D3DBLEND_INVBLENDFACTOR, IDM_ALPHA_DEST_BLENDFACTOR_INVBLENDFACTOR
1532     );
1533     return find_id(items, NUM_OF(items), value);
1534 }
1535
1536 void
1537 CMyD3DApplication::update_blending(HMENU menu, bool checked)
1538 {
1539     if (! checked)
1540         {
1541             // enable all the menu items first, so that when we're called
1542             // again with checked==false, we can disable only those that
1543             // need to be disabled.
1544             const DWORD ids[] =
1545             {
1546                 IDM_COLOR_COMPOSITE_CLEAR,
1547                 IDM_COLOR_COMPOSITE_SRC,
1548                 IDM_COLOR_COMPOSITE_DEST,
1549                 IDM_COLOR_COMPOSITE_SRC_OVER_DEST,
1550                 IDM_COLOR_COMPOSITE_DEST_OVER_SRC,
1551                 IDM_COLOR_COMPOSITE_SRC_IN_DEST,
1552                 IDM_COLOR_COMPOSITE_DEST_IN_SRC,
1553                 IDM_COLOR_COMPOSITE_SRC_ATOP_DEST,
CHAPTER 14. THE FRAME BUFFER

IDM_COLOR_COMPOSITE_DEST_ATOP_SRC,
IDM_COLOR_COMPOSITE_SRC_OUT_DEST,
IDM_COLOR_COMPOSITE_DEST_OUT_SRC,
IDM_COLOR_COMPOSITE_SRC_XOR_DEST,
IDM_COLOR_BLEND_OP_ADD,
IDM_COLOR_BLEND_OP_SUBTRACT,
IDM_COLOR_BLEND_OP_REVERSE_SUBTRACT,
IDM_COLOR_BLEND_OP_MINIMUM,
IDM_COLOR_BLEND_OP_MAXIMUM,
IDM_COLOR_SRC_BLEND_ZERO,
IDM_COLOR_SRC_BLEND_ONE,
IDM_COLOR_SRC_BLEND_SRC_COLOR,
IDM_COLOR_SRC_BLEND_INV_SRC_COLOR,
IDM_COLOR_SRC_BLEND_SRC_ALPHA,
IDM_COLOR_SRC_BLEND_INV_SRC_ALPHA,
IDM_COLOR_SRC_BLEND_DEST_ALPHA,
IDM_COLOR_SRC_BLEND_INV_DEST_ALPHA,
IDM_COLOR_SRC_BLEND_DEST_COLOR,
IDM_COLOR_SRC_BLEND_INV_DEST_COLOR,
IDM_COLOR_SRC_BLEND_SRC_ALPHA_SAT,
IDM_COLOR_SRC_BLEND_BOTH_INV_SRC_ALPHA,
IDM_COLOR_SRC_BLEND_FACTOR,
IDM_COLOR_SRC_BLEND_INV_FACTOR,
IDM_COLOR_DEST_BLEND_ZERO,
IDM_COLOR_DEST_BLEND_ONE,
IDM_COLOR_DEST_BLEND_SRC_COLOR,
IDM_COLOR_DEST_BLEND_INV_SRC_COLOR,
IDM_COLOR_DEST_BLEND_SRC_ALPHA,
IDM_COLOR_DEST_BLEND_INV_SRC_ALPHA,
IDM_COLOR_DEST_BLEND_DEST_ALPHA,
IDM_COLOR_DEST_BLEND_INV_DEST_ALPHA,
IDM_COLOR_DEST_BLEND_DEST_COLOR,
IDM_COLOR_DEST_BLEND_INV_DEST_COLOR,
IDM_COLOR_DEST_BLEND_SRC_ALPHA_SAT,
IDM_COLOR_DEST_BLEND_BOTH_INV_SRC_ALPHA,
IDM_COLOR_DEST_BLEND_FACTOR,
IDM_COLOR_DEST_BLEND_INV_FACTOR,
IDM_ALPHA_COMPOSITE_CLEAR,
IDM_ALPHA_COMPOSITE_SRC,
IDM_ALPHA_COMPOSITE_DEST,
IDM_ALPHA_COMPOSITE_SRC_OVER_DEST,
IDM_ALPHA_COMPOSITE_DEST_OVER_SRC,
IDM_ALPHA_COMPOSITE_SRC_IN_DEST,
IDM_ALPHA_COMPOSITE_DEST_IN_SRC,
IDM_ALPHA_COMPOSITE_SRC_ATOP_DEST,
IDM_ALPHA_COMPOSITE_DEST_ATOP_SRC,
IDM_ALPHA_COMPOSITE_SRC_OUT_DEST,
IDM_ALPHA_COMPOSITE_DEST_OUT_SRC,
IDM_ALPHA_COMPOSITE_SRC_XOR_DEST,
IDM_ALPHA_BLEND_OP_ADD,
IDM_ALPHA_BLEND_OP_SUBTRACT,
IDM_ALPHA_BLEND_OP_REVERSE_SUBTRACT,
IDM_ALPHA_BLEND_OP_MINIMUM,
IDM_ALPHA_BLEND_OP_MAXIMUM,
IDM_ALPHA_SRC_BLEND_ZERO,
IDM_ALPHA_SRC_BLEND_ONE,
IDM_ALPHA_SRC_BLEND_SRC_COLOR,
IDM_ALPHA_SRC_BLEND_INV_SRC_COLOR,
IDM_ALPHA_SRC_BLEND_SRC_ALPHA,
IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA,
IDM_ALPHA_SRC_BLEND_DEST_ALPHA,
IDM_ALPHA_SRC_BLEND_INV_DEST_ALPHA,
IDM_ALPHA_SRC_BLEND_DEST_COLOR,
IDM_ALPHA_SRC_BLEND_INV_DEST_COLOR,
IDM_ALPHA_SRC_BLEND_SRC_ALPHA_SAT,
IDM_ALPHA_SRC_BLEND_BOTH_INV_SRC_ALPHA,
IDM_ALPHA_SRC_BLEND_FACTOR,
IDM_ALPHA_SRC_BLEND_INV_FACTOR,
IDM_ALPHA_DEST_BLEND_ZERO,
IDM_ALPHA_DEST_BLEND_ONE,
IDM_ALPHA_DEST_BLEND_SRC_COLOR,
IDM_ALPHA_DEST_BLEND_INV_SRC_COLOR,
IDM_ALPHA_DEST_BLEND_SRC_ALPHA,
IDM_ALPHA_DEST_BLEND_INV_SRC_ALPHA,
IDM_ALPHA_DEST_BLEND_DEST_ALPHA,
IDM_ALPHA_DEST_BLEND_INV_DEST_ALPHA,
IDM_ALPHA_DEST_BLEND_DEST_COLOR,
IDM_ALPHA_DEST_BLEND_INV_DEST_COLOR,
IDM_ALPHA_DEST_BLEND_SRC_ALPHA_SAT,
IDM_ALPHA_DEST_BLEND_BOTH_INV_SRC_ALPHA,
IDM_ALPHA_DEST_BLEND_FACTOR,
IDM_ALPHA_DEST_BLEND_INV_FACTOR
};
for (UINT i = 0; i < NUM_OF(ids); i++)
{
    rt::enable_menu(menu, ids[i], true);
}
else {
    // if we don't have destination alpha, we might have to change the
    // blending selection
    if (!m_destination_alpha) {
        if (!m_color_allow_dest_alpha) {
            switch (m_color_composite) {
                case E_COMPOSITE_DEST:
                case E_COMPOSITE_DEST_OVER_SRC:
                case E_COMPOSITE_SRC_IN_DEST:
                case E_COMPOSITE_SRC_ATOP_DEST:
                case E_COMPOSITE_SRC_OUT_DEST:
                case E_COMPOSITE_SRC_XOR_DEST:
                    m_color_composite = E_COMPOSITE_NONE;
                    m_color_src_blend = D3DBLEND_ONE;
                    m_color_dest_blend = D3DBLEND_ZERO;
                    break;
            }
            switch (m_color_src_blend) {
                case D3DBLEND_DESTALPHA:
                case D3DBLEND_INVDESTALPHA:
                case D3DBLEND_DESTCOLOR:
                case D3DBLEND_INVDESTCOLOR:
                    m_color_src_blend = D3DBLEND_ONE;
                    break;
            }
            switch (m_color_dest_blend) {
                case D3DBLEND_DESTALPHA:
                case D3DBLEND_INVDESTALPHA:
                case D3DBLEND_DESTCOLOR:
                case D3DBLEND_INVDESTCOLOR:
                    m_color_dest_blend = D3DBLEND_ZERO;
                    break;
            }
        }
        if (!m_alpha_allow_dest_alpha) {
            switch (m_alpha_composite) {
            }
case E_COMPOSITE_DEST:
    case E_COMPOSITE_DEST_OVER_SRC:
    case E_COMPOSITE_SRC_IN_DEST:
    case E_COMPOSITE_SRC_ATOP_DEST:
    case E_COMPOSITE_DEST_ATOP_SRC:
    case E_COMPOSITE_SRC_OUT_DEST:
    case E_COMPOSITE_SRC_XOR_DEST:
        m_alpha_composite = E_COMPOSITE_NONE;
        m_alpha_src_blend = D3DBLEND_ONE;
        m_alpha_dest_blend = D3DBLEND_ZERO;
        break;
    }
switch (m_alpha_src_blend)
{
    case D3DBLEND_DESTALPHA:
    case D3DBLEND_INVDESTALPHA:
    case D3DBLEND_DESTCOLOR:
    case D3DBLEND_INVDESTCOLOR:
        m_alpha_src_blend = D3DBLEND_ONE;
        break;
    }
switch (m_alpha_dest_blend)
{
    case D3DBLEND_DESTALPHA:
    case D3DBLEND_INVDESTALPHA:
    case D3DBLEND_DESTCOLOR:
    case D3DBLEND_INVDESTCOLOR:
        m_alpha_dest_blend = D3DBLEND_ZERO;
        break;
    }

    // check for available blend operators
    {
        const bool can_blend_op =
            0 != (m_d3dCaps.PrimitiveMiscCaps & D3DPMISCCAPS_BLENDOP);
        rt::enable_menu(menu, IDM_COLOR_BLEND_OP_SUBTRACT, can_blend_op);
        rt::enable_menu(menu, IDM_COLOR_BLEND_OP_REVERSE_SUBTRACT, can_blend_op);
        rt::enable_menu(menu, IDM_COLOR_BLEND_OP_MINIMUM, can_blend_op);
        rt::enable_menu(menu, IDM_COLOR_BLEND_OP_MAXIMUM, can_blend_op);
        if (!can_blend_op)
            {
                if (m_color_blend_op != D3DBLENDOP_ADD)
                    {
                        m_color_blend_op = D3DBLENDOP_ADD;
                    }
if (m_alpha_blend_op != D3DBLENDOP_ADD)
{
    m_alpha_blend_op = D3DBLENDOP_ADD;
}
}

// check source and destination blend factors
const DWORD blend_caps[] =
{
    D3DPBLENDCAPS_ZERO,
    D3DPBLENDCAPS_ONE,
    D3DPBLENDCAPS_SRCCOLOR,
    D3DPBLENDCAPS_INVSRCOLOR,
    D3DPBLENDCAPS_SRCCOLOR,
    D3DPBLENDCAPS_INVSRCALPHA,
    D3DPBLENDCAPS_DESTALPHA,
    D3DPBLENDCAPS_INVDESTALPHA,
    D3DPBLENDCAPS_DESTCOLOR,
    D3DPBLENDCAPS_INVDESTCOLOR,
    D3DPBLENDCAPS_SRALPHASAT,
    0,
    D3DPBLENDCAPS_BLENDFACTOR,
    D3DPBLENDCAPS_BLENDFACTOR
};

UINT i;
for (i = 0; i < NUM_OF(blend_caps); i++)
{
    if (!blend_caps[i])
    {
        // D3DPBLENDCAPS_BOTHSRCCOLOR is obsolete, so skip it.
        continue;
    }
    bool support = 0 != (m_d3dCaps.SrcBlendCaps & blend_caps[i]);
    if (!support)
    {
        if (D3DBLEND(i+1) == m_color_src_blend)
        {
            m_color_src_blend = D3DBLEND_ONE;
        }
        if (D3DBLEND(i+1) == m_alpha_src_blend)
        {
            m_alpha_src_blend = D3DBLEND_ONE;
        }
    }
RT_FRAMEBUFFER SAMPLE APPLICATION

1783  }
1784  rt::enable_menu(menu, color_src_blend_id(D3DBLEND(i+1)), support);
1785  rt::enable_menu(menu, alpha_src_blend_id(D3DBLEND(i+1)), support);
1786
1787  support = 0 != (m_d3dCaps.DestBlendCaps & blend_caps[i]);
1788  if (!support)
1789    {
1790      if (D3DBLEND(i+1) == m_color_dest_blend)
1791        { m_color_dest_blend = D3DBLEND_ZERO;
1792       }
1793      if (D3DBLEND(i+1) == m_alpha_dest_blend)
1794        { m_alpha_dest_blend = D3DBLEND_ZERO;
1795       }
1796    }
1797  }
1798  if (D3DBLEND_BOTHINVSRCALPHA != D3DBLEND(i+1))
1799    {
1800      rt::enable_menu(menu, color_dest_blend_id(D3DBLEND(i+1)), support);
1801      rt::enable_menu(menu, alpha_dest_blend_id(D3DBLEND(i+1)), support);
1802    }
1803
1804  }
1805
1806  // check compositing operator support, i.e. combinations of blend modes
1807  for (i = 0; i < NUM_OF(sm_blend_factors); i++)
1808    {
1809      // blend factors used in this compositing operator are not supported
1810      bool support =
1811        (m_d3dCaps.SrcBlendCaps & blend_caps[sm_blend_factors[i].m_src]) &
1812        (m_d3dCaps.DestBlendCaps & blend_caps[sm_blend_factors[i].m_dest]);
1813      if (!support)
1814        {
1815          if (m_color_composite == e_composite_operator(i))
1816            {
1817              // change the blending to none
1818              m_color_composite = E_COMPOSITE_NONE;
1819            }
1820          if (m_alpha_composite == e_composite_operator(i))
1821            {
1822              m_alpha_composite = E_COMPOSITE_NONE;
1823            }
1824        }
1825
1826      // disable menu item for compositing operator i
1827      rt::enable_menu(menu, color_composite_id(e_composite_operator(i)),
1828        support);
\texttt{rt::enable_menu(menu, alpha_composite_id(e_composite_operator(i)), 1829  
  support); 1830  
} 1831  
1832  
// toggle menu items based on destination alpha supported 1833  if (!m_destination_alpha) 1834  { 1835    if (!m_color_allow_dest_alpha) 1836      { 1837        const DWORD ids[] = 1838          { 1839            IDM_COLOR_COMPOSITE_DEST, 1840            IDM_COLOR_COMPOSITE_DEST_OVER_SRC, 1841            IDM_COLOR_COMPOSITE_SRC_IN_DEST, 1842            IDM_COLOR_COMPOSITE_SRC_ATOP_DEST, 1843            IDM_COLOR_COMPOSITE_SRC_ATOP_SRC, 1844            IDM_COLOR_COMPOSITE_SRC_OUT_DEST, 1845            IDM_COLOR_SRC_BLEND_DEST_ALPHA, 1846            IDM_COLOR_SRC_BLEND_INV_DEST_ALPHA, 1847            IDM_COLOR_SRC_BLEND_DEST_COLOR, 1848            IDM_COLOR_SRC_BLEND_INV_DEST_COLOR, 1849            IDM_COLOR_DEST_BLEND_DEST_ALPHA, 1850            IDM_COLOR_DEST_BLEND_INV_DEST_ALPHA, 1851            IDM_COLOR_DEST_BLEND_DEST_COLOR, 1852            IDM_COLOR_DEST_BLEND_INV_DEST_COLOR 1853          }; 1854          for (UINT i = 0; i < NUM_OF(ids); i++) 1855            { 1856              rt::enable_menu(menu, ids[i], false); 1857            } 1858          } 1859        if (!m_alpha_allow_dest_alpha) 1860          { 1861            const DWORD ids[] = 1862              { 1863                IDM_ALPHA_COMPOSITE_DEST, 1864                IDM_ALPHA_COMPOSITE_DEST_OVER_SRC, 1865                IDM_ALPHA_COMPOSITE_SRC_IN_DEST, 1866                IDM_ALPHA_COMPOSITE_SRC_ATOP_DEST, 1867                IDM_ALPHA_COMPOSITE_SRC_ATOP_SRC, 1868                IDM_ALPHA_COMPOSITE_SRC_OUT_DEST, 1869                IDM_ALPHA_SRC_BLEND_DEST_ALPHA, 1870                IDM_ALPHA_SRC_BLEND_INV_DEST_ALPHA, 1871                IDM_ALPHA_SRC_BLEND_DEST_COLOR, 1872                IDM_ALPHA_SRC_BLEND_INV_DEST_COLOR, 1873                IDM_ALPHA_DEST_BLEND_DEST_ALPHA, 1874                IDM_ALPHA_DEST_BLEND_DEST_COLOR, 1875            }; 1876            for (UINT i = 0; i < NUM_OF(ids); i++) 1877              { 1878                rt::enable_menu(menu, ids[i], false); 1879              } 1880          } 1881      }
for (UINT i = 0; i < NUM_OF(ids); i++)
{
    rt::enable_menu(menu, ids[i], false);
}
}

// the individual source and destination blend factors are not used with min and max blend operators
if ((D3DBLENDOP_MIN == m_color_blend_op) ||
    (D3DBLENDOP_MAX == m_color_blend_op))
{
    DWORD ids[] =
    {
        IDM_COLOR_COMPOSITE_CLEAR,
        IDM_COLOR_COMPOSITE_SRC,
        IDM_COLOR_COMPOSITE_DEST,
        IDM_COLOR_COMPOSITE_SRC_OVER_DEST,
        IDM_COLOR_COMPOSITE_DEST_OVER_SRC,
        IDM_COLOR_COMPOSITE_SRC_IN_DEST,
        IDM_COLOR_COMPOSITE_DEST_IN_SRC,
        IDM_COLOR_COMPOSITE_SRC_ATOP_DEST,
        IDM_COLOR_COMPOSITE_DEST_ATOP_SRC,
        IDM_COLOR_COMPOSITE_SRC_OUT_DEST,
        IDM_COLOR_COMPOSITE_DEST_OUT_SRC,
        IDM_COLOR_COMPOSITE_SRC_XOR_DEST,
        IDM_COLOR_SRC_BLEND_ZERO,
        IDM_COLOR_SRC_BLEND_SRC_COLOR,
        IDM_COLOR_SRC_BLEND_INV_SRC_COLOR,
        IDM_COLOR_SRC_BLEND_SRC_ALPHA,
        IDM_COLOR_SRC_BLEND_INV_SRC_ALPHA,
        IDM_COLOR_SRC_BLEND_DEST_ALPHA,
        IDM_COLOR_SRC_BLEND_INV_DEST_ALPHA,
        IDM_COLOR_SRC_BLEND_DEST_COLOR,
        IDM_COLOR_SRC_BLEND_INV_DEST_COLOR,
        IDM_COLOR_SRC_BLEND_SRC_ALPHA_SAT,
        IDM_COLOR_SRC_BLEND_BOTH_INV_SRC_ALPHA,
        IDM_COLOR_DEST_BLEND_ONE,
        IDM_COLOR_DEST_BLEND_SRC_COLOR,
        IDM_COLOR_DEST_BLEND_INV_SRC_COLOR,
        IDM_COLOR_DEST_BLEND_SRC_ALPHA,
        IDM_COLOR_DEST_BLEND_INV_SRC_ALPHA,
CHAPTER 14. THE FRAME BUFFER

1921 IDM_COLOR_DEST_BLEND_DEST_ALPHA,
1922 IDM_COLOR_DEST_BLEND_INV_DEST_ALPHA,
1923 IDM_COLOR_DEST_BLEND_DEST_COLOR,
1924 IDM_COLOR_DEST_BLEND_INV_DEST_COLOR
1925 }
1926 for (i = 0; i < NUM_OF(ids); i++)
1927 {
1928     rt::enable_menu(menu, ids[i], false);
1929 }
1930 m_color_composite = E_COMPOSITE_NONE;
1931 m_color_src_blend = D3DBLEND_ONE;
1932 m_color_dest_blend = D3DBLEND_ZERO;
1933 }
1934 if ((D3DBLENDOP_MIN == m_alpha_blend_op) ||
1935     (D3DBLENDOP_MAX == m_alpha_blend_op))
1936 {
1937     DWORD ids[] =
1938     {
1939         IDM_ALPHA_COMPOSITE_CLEAR,
1940         IDM_ALPHA_COMPOSITE_SRC,
1941         IDM_ALPHA_COMPOSITE_DEST,
1942         IDM_ALPHA_COMPOSITE_SRC_OVER_DEST,
1943         IDM_ALPHA_COMPOSITE_DEST_OVER_SRC,
1944         IDM_ALPHA_COMPOSITE_SRC_IN_DEST,
1945         IDM_ALPHA_COMPOSITE_DEST_IN_SRC,
1946         IDM_ALPHA_COMPOSITE_SRC_ATOP_DEST,
1947         IDM_ALPHA_COMPOSITE_DEST_ATOP_SRC,
1948         IDM_ALPHA_COMPOSITE_SRC_OUT_DEST,
1949         IDM_ALPHA_COMPOSITE_DEST_OUT_SRC,
1950         IDM_ALPHA_COMPOSITE_SRC_XOR_DEST,
1951         IDM_ALPHA_SRC_BLEND_ZERO,
1952         IDM_ALPHA_SRC_BLEND_SRC_COLOR,
1953         IDM_ALPHA_SRC_BLEND_SRC_ALPHA,
1954         IDM_ALPHA_SRC_BLEND_SRC_ALPHA_SAT,
1955         IDM_ALPHA_SRC_BLEND_DEST_ALPHA,
1956         IDM_ALPHA_SRC_BLEND_DEST_COLOR,
1957         IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA,
1958         IDM_ALPHA_SRC_BLEND_INV_SRC_COLOR,
1959         IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA_SAT,
1960         IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA,
1961         IDM_ALPHA_SRC_BLEND_INV_SRC_COLOR,
1962         IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA,
1963         IDM_ALPHA_SRC_BLEND_INV_SRC_COLOR,
1964         IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA,
1965         IDM_ALPHA_SRC_BLEND_INV_SRC_COLOR,
1966         IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA,

for (i = 0; i < NUM_OF(ids); i++)
{
    rt::enable_menu(menu, ids[i], false);
}
m_alpha_composite = E_COMPOSITE_NONE;
m_alpha_src_blend = D3DBLEND_ONE;
m_alpha_dest_blend = D3DBLEND_ZERO;
}

// now patch up m_color_composite in case it is inconsistent with
// m_color_src_blend and m_color_dest_blend
for (i = 0; i < NUM_OF(sm_blend_factors); i++)
{
    if ((sm_blend_factors[i].m_src == m_color_src_blend) &&
        (sm_blend_factors[i].m_dest == m_color_dest_blend))
    {
        m_color_composite = e_composite_operator(i);
        break;
    }
}
if (NUM_OF(sm_blend_factors) == i)
{
    m_color_composite = E_COMPOSITE_NONE;
}
for (i = 0; i < NUM_OF(sm_blend_factors); i++)
{
    if ((sm_blend_factors[i].m_src == m_alpha_src_blend) &&
        (sm_blend_factors[i].m_dest == m_alpha_dest_blend))
    {
        m_alpha_composite = e_composite_operator(i);
        break;
    }
}
if (NUM_OF(sm_blend_factors) == i)
{
    m_alpha_composite = E_COMPOSITE_NONE;
}
rt::check_menu(menu, color_blend_op_id(m_color_blend_op), checked);
rt::check_menu(menu, alpha_blend_op_id(m_alpha_blend_op), checked);
if (E_COMPOSITE_NONE != m_color_composite)
{
    rt::check_menu(menu, color_composite_id(m_color_composite), checked);
}
rt::check_menu(menu, color_src_blend_id(m_color_src_blend), checked);
rt::check_menu(menu, color_dest_blend_id(m_color_dest_blend), checked);
if (E_COMPOSITE_NONE != m_alpha_composite)
{
    rt::check_menu(menu, alpha_composite_id(m_alpha_composite), checked);
}
rt::check_menu(menu, alpha_src_blend_id(m_alpha_src_blend), checked);
rt::check_menu(menu, alpha_dest_blend_id(m_alpha_dest_blend), checked);

// disable blending submenus if blending is disabled
rt::enable_menu(menu, IDM_COLOR_BLEND_DEST_ALPHA, m_color_blend_enable);
able_menu(menu, _T("&Color Blend"), m_color_blend_enable);
TWS(::DrawMenuBar(m_hWnd));
HMENU blend_menu = TWS(find_menu(menu, _T("&Color Blend")));
enable_menu(blend_menu, _T("&Operator"), m_color_blend_enable);
enable_menu(blend_menu, _T("&Compositing"), m_color_blend_enable);
enable_menu(blend_menu, _T("&Source"), m_color_blend_enable);
enable_menu(blend_menu, _T("&Destination"), m_color_blend_enable);
rt::enable_menu(menu, IDM_ALPHA_BLEND_DEST_ALPHA, m_alpha_blend_enable);
blend_menu = TWS(find_menu(menu, _T("&Alpha Blend")));
enable_menu(blend_menu, _T("&Operator"), m_alpha_blend_enable);
enable_menu(blend_menu, _T("&Compositing"), m_alpha_blend_enable);
enable_menu(blend_menu, _T("&Source"), m_alpha_blend_enable);
enable_menu(blend_menu, _T("&Destination"), m_alpha_blend_enable);
}
switch (control)
{
    #define TOGGLE_COLOR_WRITE_ENABLE(id_, state_)
    case IDM_COLOR_WRITE_ENABLE_##id_: \
        rt::toggle_menu(menu, IDM_COLOR_WRITE_ENABLE_##id_, state_); \
        set_color_write_enable(); \
        handled = true; \
        break
    TOGGLE_COLOR_WRITE_ENABLE(RED, m_color_write_enable_red);
    TOGGLE_COLOR_WRITE_ENABLE(GREEN, m_color_write_enable_green);
    TOGGLE_COLOR_WRITE_ENABLE(BLUE, m_color_write_enable_blue);
    TOGGLE_COLOR_WRITE_ENABLE(ALPHA, m_color_write_enable_alpha);
    #undef TOGGLE_COLOR_WRITE_ENABLE

    #define TOGGLE(id_, state_)
    case id_: \
        rt::toggle_menu(menu, id_, state_); \
        handled = true; \
        break
    TOGGLE(IDM_Z_WRITE_ENABLE, m_z_write_enable);
    TOGGLE(IDM_ANIMATE_VIEW, m_animate_view);
    TOGGLE(IDM_SHOW_STATS, m_show_stats);
    TOGGLE(IDM_OPTION_DITHER, m_dithered);
    TOGGLE(IDM_OPTION_BACKGROUND_TILE, m_tile_background);
    #undef TOGGLE

    #define MODIFY_COLOR(id_, state_, transparent_)
    case id_: \
    { \n        rt::pauser block(*this); \
        state_ = rt::choose_color_transparent(window, state_, transparent_); \
    } \
    handled = true; \
    break
    MODIFY_COLOR(IDM_EDIT_DIFFUSE_COLOR, m_fg, true);
    MODIFY_COLOR(IDM_EDIT_TEXT_COLOR, m_text_fg, true);
    MODIFY_COLOR(IDM_EDIT_SPECULAR_COLOR, m_specular, false);
    #undef MODIFY_COLOR

    case IDM_EDIT_BLEND_FACTOR:
    { 
        rt::pauser block(*this);
        m_blend_factor = rt::choose_color(window, m_blend_factor);
    } \
    handled = true; \
    break;
case IDM_EDIT_BG_COLOR:
    {
        rt::pauser block(*this);
        m_bg = rt::choose_color(window, m_bg);
    }
    handled = true;
    break;

case IDM_TEXTURED:
    m_textured = m_can_texture && !m_textured;
    rt::check_menu(menu, IDM_TEXTURED, m_textured);
    handled = true;
    break;

case IDM_FILE_OPEN_TEXTURE:
    {
        rt::pauser block(*this);
        on_texture_file();
    }
    handled = true;
    break;

case IDM_STENCIL_IRREGULAR_MASK:
    rt::toggle_menu(menu, IDM_STENCIL_IRREGULAR_MASK, m_stencil_mask);
    m_build_stencil = true;
    handled = true;
    break;

case IDM_STENCIL_STIPPLING:
    rt::toggle_menu(menu, IDM_STENCIL_STIPPLING, m_stencil_stippling);
    m_build_stencil = true;
    handled = true;
    break;

case IDM_MULTISAMPLE_ANTIALIAS:
    rt::toggle_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, m_multisample_antialias);
    m_multisample_depth_of_field = false;
    m_multisample_motion_blur = false;
    rt::check_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, m_multisample_depth_of_field);
    rt::check_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, m_multisample_motion_blur);
    handled = true;
    break;

case IDM_MULTISAMPLE_DEPTH_OF_FIELD:
    rt::toggle_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, m_multisample_depth_of_field);
    handled = true;
    break;
m_multisample_antialias = false;
m_multisample_motion_blur = false;
rt::check_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, m_multisample_antialias);
rt::check_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, m_multisample_motion_blur);
handled = true;
break;

case IDM_MULTISAMPLE_MOTION_BLUR:
  rt::toggle_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, m_multisample_motion_blur);
m_multisample_antialias = false;
m_multisample_depth_of_field = false;
rt::check_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, m_multisample_antialias);
rt::check_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, m_multisample_depth_of_field)
handled = true;
break;

case IDM_COLOR_SRC_BLEND_ZERO:
case IDM_COLOR_SRC_BLEND_ONE:
case IDM_COLOR_SRC_BLEND_SRC_COLOR:
case IDM_COLOR_SRC_BLEND_INV_SRC_COLOR:
case IDM_COLOR_SRC_BLEND_SRC_ALPHA:
case IDM_COLOR_SRC_BLEND_INV_SRC_ALPHA:
case IDM_COLOR_SRC_BLEND_DEST_ALPHA:
case IDM_COLOR_SRC_BLEND_INV_DEST_ALPHA:
case IDM_COLOR_SRC_BLEND_DEST_COLOR:
case IDM_COLOR_SRC_BLEND_INV_DEST_COLOR:
case IDM_COLOR_SRC_BLEND_SRC_ALPHA_SAT:
case IDM_COLOR_SRC_BLEND_BOTH_INV_SRC_ALPHA:
case IDM_COLOR_SRC_BLEND_FACTOR:
case IDM_COLOR_SRC_BLEND_INV_FACTOR:
  update_blending(menu, false);
m_color_src_blend = D3DBLEND(get_menu_data(control));
  update_blending(menu, true);
handled = true;
break;

m_color_dest_blend = D3DBLEND(get_menu_data(control));
update_blending(menu, false);
update_blending(menu, true);
handled = true;
break;
case IDM_COLOR_DEST_BLEND_SRC_ALPHA_SAT:
    update_blending(menu, false);
    m_color_dest_blend = D3DBLEND(get_menu_data(control));
    update_blending(menu, true);
    handled = true;
    break;

case IDM_COLOR_DEST_BLEND_FACTOR:
    case IDM_COLOR_DEST_BLEND_INV_FACTOR:
        update_blending(menu, false);
        m_color_dest_blend = sm_blend_factors[m_color_dest_blend].m_src;
        update_blending(menu, true);
        handled = true;
        break;

    case IDM_COLOR_COMPOSITE_CLEAR:
        case IDM_COLOR_COMPOSITE_SRC:
            case IDM_COLOR_COMPOSITE_DEST:
                case IDM_COLOR_COMPOSITE_SRC_OVER_DEST:
                    case IDM_COLOR_COMPOSITE_DEST_OVER_SRC:
                        case IDM_COLOR_COMPOSITE_SRC_IN_DEST:
                            case IDM_COLOR_COMPOSITE_DEST_IN_SRC:
                                case IDM_COLOR_COMPOSITE_SRC_ATOP_DEST:
                                    case IDM_COLOR_COMPOSITE_DEST_ATOP_SRC:
                                        case IDM_COLOR_COMPOSITE_SRC_OUT_DEST:
                                            case IDM_COLOR_COMPOSITE_DEST_OUT_SRC:
                                                case IDM_COLOR_COMPOSITE_SRC_XOR_DEST:
                                                    update_blending(menu, false);
                                                    m_color_composite = e_composite_operator(get_menu_data(control));
                                                    m_color_src_blend = sm_blend_factors[m_color_composite].m_src;
                                                    m_color_dest_blend = sm_blend_factors[m_color_composite].m_dest;
                                                    update_blending(menu, true);
                                                    handled = true;
                                                    break;

    case IDM_COLOR_BLEND_DEST_ALPHA:
        update_blending(menu, false);
        rt::toggle_menu(menu, IDM_COLOR_BLEND_DEST_ALPHA, m_color_allow_dest_alpha);
        update_blending(menu, true);
        handled = true;
        break;

    case IDM_COLOR_BLEND_ENABLE:
        update_blending(menu, false);
        m_color_blend_enable = !m_color_blend_enable;
        update_blending(menu, true);
        handled = true;
        break;

    case IDM_COLOR_BLEND_OP_ADD:
        case IDM_COLOR_BLEND_OP_SUBTRACT:
            case IDM_COLOR_BLEND_OP_REVERSE_SUBTRACT:
case IDM_COLOR_BLEND_OP_MINIMUM:
    update_blending(menu, false);
    m_color_blend_op = D3DBLENDOP(get_menu_data(control));
    update_blending(menu, true);
    handled = true;
    break;

    case IDM_ALPHA_SRC_BLEND_ZERO:
    case IDM_ALPHA_SRC_BLEND_ONE:
    case IDM_ALPHA_SRC_BLEND_SRC_COLOR:
    case IDM_ALPHA_SRC_BLEND_INV_SRC_COLOR:
    case IDM_ALPHA_SRC_BLEND_SRC_ALPHA:
    case IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA:
    case IDM_ALPHA_SRC_BLEND_DEST_ALPHA:
    case IDM_ALPHA_SRC_BLEND_INV_DEST_ALPHA:
    case IDM_ALPHA_SRC_BLEND_SRC_ALPHA_SAT:
    case IDM_ALPHA_SRC_BLEND_BOTH_INV_SRC_ALPHA:
    case IDM_ALPHA_SRC_BLEND_FACTOR:
    case IDM_ALPHA_SRC_BLEND_INV_FACTOR:
        update_blending(menu, false);
        m_alpha_src_blend = D3DBLEND(get_menu_data(control));
        update_blending(menu, true);
        handled = true;
        break;

    case IDM_ALPHA_DEST_BLEND_ZERO:
    case IDM_ALPHA_DEST_BLEND_ONE:
    case IDM_ALPHA_DEST_BLEND_SRC_COLOR:
    case IDM_ALPHA_DEST_BLEND_INV_SRC_COLOR:
    case IDM_ALPHA_DEST_BLEND_SRC_ALPHA:
    case IDM_ALPHA_DEST_BLEND_INV_SRC_ALPHA:
    case IDM_ALPHA_DEST_BLEND_DEST_ALPHA:
    case IDM_ALPHA_DEST_BLEND_INV_DEST_ALPHA:
    case IDM_ALPHA_DEST_BLEND_DEST_COLOR:
    case IDM_ALPHA_DEST_BLEND_INV_DEST_COLOR:
    case IDM_ALPHA_DEST_BLEND_SRC_ALPHA_SAT:
    case IDM_ALPHA_DEST_BLEND_FACTOR:
        update_blending(menu, false);
        m_alpha_dest_blend = D3DBLEND(get_menu_data(control));
        update_blending(menu, true);
        handled = true;
        break;
case IDM_ALPHA_COMPOSITE_CLEAR:
    break;

    case IDM_ALPHA_COMPOSITE_SRC:
    break;

    case IDM_ALPHA_COMPOSITE_DEST:
    break;

    case IDM_ALPHA_COMPOSITE_SRC_OVER_DEST:
    break;

    case IDM_ALPHA_COMPOSITE_DEST_OVER_SRC:
    break;

    case IDM_ALPHA_COMPOSITE_SRC_IN_DEST:
    break;

    case IDM_ALPHA_COMPOSITE_DEST_IN_SRC:
    break;

    case IDM_ALPHA_COMPOSITE_SRC_ATOP_DEST:
    break;

    case IDM_ALPHA_COMPOSITE_DEST_ATOP_SRC:
    break;

    case IDM_ALPHA_COMPOSITE_SRC_OUT_DEST:
    break;

    case IDM_ALPHA_COMPOSITE_DEST_OUT_SRC:
    break;

    case IDM_ALPHA_COMPOSITE_SRC_XOR_DEST:
        update_blending(menu, false);
        m_alpha_composite = e_composite_operator(get_menu_data(control));
        m_alpha_src_blend = sm_blend_factors[m_alpha_composite].m_src;
        m_alpha_dest_blend = sm_blend_factors[m_alpha_composite].m_dest;
        update_blending(menu, true);
        handled = true;
        break;

    case IDM_ALPHA_BLEND_DEST_ALPHA:
        update_blending(menu, false);
        rt::toggle_menu(menu, IDM_ALPHA_BLEND_DEST_ALPHA, m_alpha_allow_dest_alpha);
        update_blending(menu, true);
        handled = true;
        break;

    case IDM_ALPHA_BLEND_ENABLE:
        update_blending(menu, false);
        m_alpha_blend_enable = !m_alpha_blend_enable;
        update_blending(menu, true);
        handled = true;
        break;

    case IDM_ALPHA_BLEND_OP_ADD:
    case IDM_ALPHA_BLEND_OP_SUBTRACT:
    case IDM_ALPHA_BLEND_OP_REVERSE_SUBTRACT:
    case IDM_ALPHA_BLEND_OP_MINIMUM:
    case IDM_ALPHA_BLEND_OP_MAXIMUM:
        update_blending(menu, false);
        m_alpha_blend_op = D3DBLENDOP(get_menu_data(control));
        update_blending(menu, true);
        handled = true;
        break;
case IDM_RESET_VIEW:
    m_rot_x = 0.0f;
    m_rot_y = 0.0f;
    handled = true;
    break;

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

    return 0;

// CMyD3DApplication::on_texture_file
void CMyD3DApplication::on_texture_file()
{
    TCHAR file[MAX_PATH] = { 0 };
    OPENFILENAME ofn =
    {
        sizeof(ofn), 0, 0,
        _T("All images\0")
        _T("*.bmp;*.jpg;*.png;*.pbm;*.pgm;*.ppm;*.pnm;*.tga;*.tif\0")
        _T("All files (*.*)\0")
        _T("Bitmap images (*.bmp)\0")
        _T("JPEG images (*.jpg)\0")
        _T("PNG images (*.png)\0")
        _T("Portable bitmap (*.pbm)\0")
        _T("Portable grayscale (*.pgm)\0")
        _T("Portable pixmap (*.ppm)\0")
        _T("Portable anymap (*.pnm)\0")
        _T("Targa images (*.tga)\0")
        _T("TIFF images (*.tif)\0")
        "\0",
        0, 0, 0,
        file, MAX_PATH, 0, 0, 0, OFN_FILEMUSTEXIST
    };

if (::GetOpenFileName(&ofn))
{
    create_texture(file);
}

////////////////////////////////////////////////////////////
// set_color_write_enable

// Set RS Color Write Enable based on UI state.

void CMyD3DApplication::set_color_write_enable()
{
    THR(m_pd3dDevice->SetRenderState(D3DRS_COLORWRITEENABLE,
                                      (m_color_write_enable_red ? D3DCOLORWRITEENABLE_RED : 0) |
                                      (m_color_write_enable_green ? D3DCOLORWRITEENABLE_GREEN : 0) |
                                      (m_color_write_enable_blue ? D3DCOLORWRITEENABLE_BLUE : 0) |
                                      (m_color_write_enable_alpha ? D3DCOLORWRITEENABLE_ALPHA : 0)));}

////////////////////////////////////////////////////////////
// render_stencil_mask

// Use the torus mesh to render a hollow circle mask into
// the 0x1 stencil plane.

void CMyD3DApplication::render_stencil_mask()
{
    // set the reference and mask values for plane i
    rt::s_rs states[] =
    {
        D3DRS_STENCILREF, 0x1,
        D3DRS_STENCILMASK, 0x1,
        D3DRS_STENCILWRITEMASK, 0x1,
    };
    rt::set_states(m_pd3dDevice, states, NUM_OF(states));

    // disable texturing
    const rt::s_tss ts_states[] =
    {
        D3DTSS_COLOROP, D3DTOP_DISABLE,
        D3DTSS_ALPHAOP, D3DTOP_DISABLE
    };
    rt::set_states(m_pd3dDevice, 0, ts_states,
                   NUM_OF(ts_states));
THR(m_pd3dDevice->SetTexture(0, NULL));

// draw torus
D3DXMATRIX ident(1, 0, 0, 0,
0, 1, 0, 0,
0, 0, 1, 0,
0, 0, 0, 1);
THR(m_pd3dDevice->SetTransform(D3DTS_WORLD, &ident));
THR(m_torus->DrawSubset(0));
}

////////////////////////////////////////////////////////////
// render_stencil_stipple
// Render a stipple pattern into the 0x2 stencil plane.
// void CMyD3DApplication:render_stencil_stipple()
{
    // set the reference and mask values for plane 2; the
    // alpha test rejects stipple texels with zero alpha,
    // so that the stencil bits are not written there.
    rt::s_rs states[] =
    {
        D3DRS_ALPHATESTENABLE, true,
        D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL,
        D3DRS_ALPHAREF, 0x1,
        D3DRS_STENCILREF, 0x2,
        D3DRS_STENCILMASK, 0x2,
        D3DRS_STENCILWRITEMASK, 0x2,
        D3DRS_COLORVERTEX, true,
        D3DRS_LIGHTING, false,
        D3DRS_SPECULARENABLE, false
    };
    rt::set_states(m_pd3dDevice, states, NUM_OF(states));
    rt::s_tss tex_states[] =
    {
        D3DTSS_COLOROP, D3DTOP_SELECTARG1,
        D3DTSS_COLORARG1, D3DTA_TEXTURE,
        D3DTSS_COLORARG2, D3DTA_DIFFUSE,
        D3DTSS_ALPHAOP, D3DTOP_SELECTARG1,
        D3DTSS_ALPHARG1, D3DTA_TEXTURE,
        D3DTSS_ALPHARG2, D3DTA_DIFFUSE
    };
    rt::set_states(m_pd3dDevice, 0,
    tex_states, NUM_OF(tex_states));
THR(m_pd3dDevice->SetTexture(0, m_stipple));

// draw stipple with screen-space triangles that cover
// the whole screen
THR(m_pd3dDevice->SetFVF(s_screen_vertex::FVF));
THR(m_pd3dDevice->SetStreamSource(0, m_stipple_verts, 0,
    sizeof(s_screen_vertex)));
THR(m_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0,
    m_num_stipple_quads*2));
}

////////////////////////////////////////////////////////////
// render_stencil

// Build the stencil plane(s) depending on UI state.
void CMyD3DApplication::render_stencil()
{
    // clear the stencil planes first
    THR(m_pd3dDevice->Clear(0L, NULL, D3DCLEAR_STENCIL,
        0, 1.0f, 0L));

    // set states for stencil rendering
    rt::s_rs states[] =
    {
        D3DRS_STENCILENABLE, true,
        D3DRS_STENCILFUNC, D3DCMP_ALWAYS,
        D3DRS_STENCILPASS, D3DSTENCILOP_REPLACE,
        D3DRS_STENCILFAIL, D3DSTENCILOP_REPLACE,
        D3DRS_STENCILZFAIL, D3DSTENCILOP_REPLACE,
        D3DRS_COLORWRITEENABLE, 0L,
        D3DRS_ALPHABLENDENABLE, m_color_blend_enable,
        D3DRS_SRCBLEND, D3DBLEND_ZERO,
        D3DRS_DESTBLEND, D3DBLEND_ONE
    };
    rt::set_states(m_pd3dDevice, states, NUM_OF(states));

    // render geometry to set the planes
    if (m_stencil_mask)
    {
        render_stencil_mask();
    }
    if (m_stencil_stippling)
    {
        render_stencil_stipple();
    }
// set the state for masking against stencil bits
rt::s_rs use_stencil_states[] =
{
    D3DRS_STENCILENABLE, true,
    D3DRS_STENCILFUNC, D3DCMP_EQUAL,
    D3DRS_STENCILREF, 0,
    D3DRS_STENCILMASK, ~0UL,
    D3DRS_STENCILWRITEMASK, 0,
    D3DRS_STENCILPASS, D3DSTENCILOP_KEEP,
    D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP,
    D3DRS_STENCILZFAIL, D3DSTENCILOP_KEEP
};
rt::set_states(m_pd3dDevice, use_stencil_states,
    NUM_OF(use_stencil_states));
set_color_write_enable();

void CMyD3DApplication::render_teapot(const D3DXMATRIX &matrix)
{
    THR(m_pd3dDevice->SetTransform(D3DTS_WORLD, &matrix));
    THR(m_teapot->DrawSubset(0));
}

UINT passes = std::max(UINT(m_d3dpp.MultiSampleType), 1U);
float rot_x = m_last_rot_x;
float delta_x = (m_rot_x - m_last_rot_x)/passes;
float rot_y = m_last_rot_y;
float delta_y = (m_rot_y - m_last_rot_y)/passes;
2565      for (UINT i = 0; i < passes; i++)
2566      {
2567          THR(m_pd3dDevice->SetRenderState(
2568              D3DRS_MULTISAMPLEMASK, 1L << i));
2569          render_teapot(rt::mat_rot_x(rot_x)*
2570              rt::mat_rot_y(rot_y));
2571          rot_x += delta_x;
2572          rot_y += delta_y;
2573      }
2574      THR(m_pd3dDevice->SetRenderState(D3DRS_MULTISAMPLEMASK,
2575          ~0UL));
2576  }
2577
2578  //////////////////////////////////////////////////////////////////////////
2579  // render_depth_of_field
2580  //
2581  // Render 9 teapots with a jittered view frustum to
2582  // create a depth of field effect.
2583  //
2584  void
2585  CMyD3DApplication::render_depth_of_field()
2586  {
2587      const D3DXMATRIX base = rt::mat_rot_x(m_rot_x)*
2588          rt::mat_rot_y(m_rot_y)*rt::mat_scale(0.5f);
2589      const UINT passes =
2590          std::max(UINT(m_d3dpp.MultiSampleType), 1U);
2591      const float *jitter = sm_jitter[passes-1];
2592      for (UINT i = 0; i < passes; i++)
2593      {
2594          THR(m_pd3dDevice->SetRenderState(
2595              D3DRS_MULTISAMPLEMASK, 1L << i));
2596          rt::mat_look_at view(D3DXVECTOR3(
2597              jitter[i*2+0]*0.25f, jitter[i*2+1]*0.25f, -5));
2598          THR(m_pd3dDevice->SetTransform(D3DTS_VIEW, &view));
2599          render_teapot(base*rt::mat_trans(-1, -1, 1));
2600          render_teapot(base*rt::mat_trans(1, -1, 1));
2601          render_teapot(base*rt::mat_trans(-1, 1, 1));
2602          render_teapot(base*rt::mat_trans(1, 1, 1));
2603          render_teapot(base);
14.10. **RT_FRAMEBUFFER SAMPLE APPLICATION**

```cpp
rt::mat_look_at view(D3DXVECTOR3(0, 0, -5));
THR(m_pd3dDevice->SetTransform(D3DTS_VIEW, &view));
THR(m_pd3dDevice->SetRenderState(D3DRS_MULTISAMPLEMASK, ~0UL));
}

/////////////////////////////////////////////////////////////////////
// CMyD3DApplication::tile_background
//
// Use CopyRects to tile the back buffer with an image containing alpha.
// The rectangle and point lists are rebuilt whenever RestoreDeviceObjects
// is called. This keeps the lists updated whenever the device changes
// or the window is resized.
//
void CMyD3DApplication::tile_background()
{
    // draw background tile, storing alpha straight into destination
    CComPtr<IDirect3DSurface9> back;
    THR(m_pd3dDevice->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &back));
    for (UINT i = 0; i < m_tile_rects.size(); i++)
    {
        RECT r =
        {
            m_tile_offsets[i].x, m_tile_offsets[i].y,
            m_tile_offsets[i].x + m_tile_rects[i].right - m_tile_rects[i].left,
            m_tile_offsets[i].y + m_tile_rects[i].bottom - m_tile_rects[i].top
        };
        THR(m_pd3dDevice->StretchRect(m_device_tile, &m_tile_rects[i],
                               back, &r, D3DTEXF_NONE));
    }
}

DWORD CMyD3DApplication::PresentFlags() const
{
    DWORD flags = CD3DApplication::PresentFlags();
    if (m_d3dEnumeration.AppMinStencilBits > 0)
    {
        flags &= ~D3DPRESENTFLAG_DISCARD_DEPTHSTENCIL;
    }
    return flags;
}
```
struct s_menu_data
{
    UINT m_id;
    DWORD m_data;
};

void CMYD3DApplication::set_menu_data()
{
    const s_menu_data data[] =
    {
        IDM_COLOR_COMPOSITE_CLEAR, E_COMPOSITE_CLEAR,
        IDM_COLOR_COMPOSITE_SRC, E_COMPOSITE_SRC,
        IDM_COLOR_COMPOSITE_DEST, E_COMPOSITE_DEST,
        IDM_COLOR_COMPOSITE_SRC_OVER_DEST, E_COMPOSITE_SRC_OVER_DEST,
        IDM_COLOR_COMPOSITE_DEST_OVER_SRC, E_COMPOSITE_DEST_OVER_SRC,
        IDM_COLOR_COMPOSITE_SRC_IN_DEST, E_COMPOSITE_SRC_IN_DEST,
        IDM_COLOR_COMPOSITE_DEST_IN_SRC, E_COMPOSITE_DEST_IN_SRC,
        IDM_COLOR_COMPOSITE_SRC_ATOP_DEST, E_COMPOSITE_SRC_ATOP_DEST,
        IDM_COLOR_COMPOSITE_DEST_ATOP_SRC, E_COMPOSITE_DEST_ATOP_SRC,
        IDM_COLOR_COMPOSITE_SRC_OUT_DEST, E_COMPOSITE_SRC_OUT_DEST,
        IDM_COLOR_COMPOSITE_DEST_OUT_SRC, E_COMPOSITE_DEST_OUT_SRC,
        IDM_COLOR_COMPOSITE_SRC_XOR_DEST, E_COMPOSITE_SRC_XOR_DEST,
        IDM_COLOR_BLEND_OP_ADD, D3DBLENDOP_ADD,
        IDM_COLOR_BLEND_OP_SUBTRACT, D3DBLENDOP_SUBTRACT,
        IDM_COLOR_BLEND_OP_REVERSE_SUBTRACT, D3DBLENDOP_REVSUBTRACT,
        IDM_COLOR_BLEND_OP_MINIMUM, D3DBLENDOP_MIN,
        IDM_COLOR_BLEND_OP_MAXIMUM, D3DBLENDOP_MAX,
        IDM_COLOR_SRC_BLEND_ZERO, D3DBLEND_ZERO,
        IDM_COLOR_SRC_BLEND_ONE, D3DBLEND_ONE,
        IDM_COLOR_SRC_BLEND_SRC_COLOR, D3DBLEND_SRCCOLOR,
        IDM_COLOR_SRC_BLEND_INV_SRC_COLOR, D3DBLEND_INVSRCCOLOR,
        IDM_COLOR_SRC_BLEND_SRC_ALPHA, D3DBLEND_SRCCALPHA,
        IDM_COLOR_SRC_BLEND_INV_SRC_ALPHA, D3DBLEND_INVSRCCALPHA,
        IDM_COLOR_SRC_BLEND_DEST_ALPHA, D3DBLEND_DESTALPHA,
        IDM_COLOR_SRC_BLEND_INV_DEST_ALPHA, D3DBLEND_INVDESTALPHA,
        IDM_COLOR_SRC_BLEND_DEST_COLOR, D3DBLEND_DESTCOLOR,
        IDM_COLOR_SRC_BLEND_INV_DEST_COLOR, D3DBLEND_INVDESTCOLOR,
        IDM_COLOR_SRC_BLEND_SRCCOLOR, D3DBLEND_SRCCOLOR,
        IDM_COLOR_SRC_BLEND_SRCCALPHA, D3DBLEND_SRCCALPHA,
        IDM_COLOR_SRC_BLEND_SRCCOLOR_SRCCALPHA, D3DBLEND_SRCCOLORSRCCALPHA,
        IDM_COLOR_SRC_BLEND_SRCCOLOR_SRCCALPHA_BOTH_INV_SRCCOLOR_SRCCALPHA, D3DBLEND_SRCCOLORSRCCALPHA_BOTHINVSRCCOLORSRCCALPHA,
IDM_COLOR_DEST_BLEND_SRC_ALPHA, D3DBLEND_SRCALPHA,
IDM_COLOR_DEST_BLEND_INV_SRC_ALPHA, D3DBLEND_INVSRCCOLOR,
IDM_COLOR_DEST_BLEND_DEST_ALPHA, D3DBLEND_DESTCOLOR,
IDM_COLOR_DEST_BLEND_INV_DEST_ALPHA, D3DBLEND_INVDESTCOLOR,
IDM_COLOR_DEST_BLEND_DEST_COLOR, D3DBLEND_DESTCOLOR,
IDM_COLOR_DEST_BLEND_INV_DEST_COLOR, D3DBLEND_INVDESTCOLOR,
IDM_COLOR_DEST_BLEND_SRC_ALPHA_SAT, D3DBLEND_SRCCOLOR,
IDM_COLOR_DEST_BLEND_BOTH_INV_SRC_ALPHA, D3DBLEND_BOTHINVSRCALPHA,
IDM_COLOR_DEST_BLEND_FACTOR, D3DBLEND_BLENDFACTOR,
IDM_COLOR_DEST_BLEND_INV_FACTOR, D3DBLEND_INVBLENDFACTOR,
IDM_ALPHA_COMPOSITE_CLEAR, E_COMPOSITE_CLEAR,
IDM_ALPHA_COMPOSITE_SRC, E_COMPOSITE_SRC,
IDM_ALPHA_COMPOSITE_DEST, E_COMPOSITE_DEST,
IDM_ALPHA_COMPOSITE_SRC_OVER_DEST, E_COMPOSITE_SRC_OVER_DEST,
IDM_ALPHA_COMPOSITE_DEST_OVER_SRC, E_COMPOSITE_DEST_OVER_SRC,
IDM_ALPHA_COMPOSITE_SRC_IN_DEST, E_COMPOSITE_SRC_IN_DEST,
IDM_ALPHA_COMPOSITE_DEST_IN_SRC, E_COMPOSITE_DEST_IN_SRC,
IDM_ALPHA_COMPOSITE_SRC_ATOP_DEST, E_COMPOSITE_SRC_ATOP_DEST,
IDM_ALPHA_COMPOSITE_DEST_ATOP_SRC, E_COMPOSITE_DEST_ATOP_SRC,
IDM_ALPHA_COMPOSITE_SRC_OUT_DEST, E_COMPOSITE_SRC_OUT_DEST,
IDM_ALPHA_COMPOSITE_DEST_OUT_SRC, E_COMPOSITE_DEST_OUT_SRC,
IDM_ALPHA_COMPOSITE_SRC_XOR_DEST, E_COMPOSITE_SRC_XOR_DEST,
IDM_ALPHA_BLEND_OP_ADD, D3DBLENDOP_ADD,
IDM_ALPHA_BLEND_OP_SUBTRACT, D3DBLENDOP_SUBTRACT,
IDM_ALPHA_BLEND_OP_REVERSE_SUBTRACT, D3DBLENDOP_REVSUBTRACT,
IDM_ALPHA_BLEND_OP_MINIMUM, D3DBLEND_MIN,
IDM_ALPHA_BLEND_OP_MAXIMUM, D3DBLEND_MAX,
IDM_ALPHA_SRC_BLEND_ZERO, D3DBLEND_ZERO,
IDM_ALPHA_SRC_BLEND_ONE, D3DBLEND_ONE,
IDM_ALPHA_SRC_BLEND_SRCCOLOR, D3DBLEND_SRCCOLOR,
IDM_ALPHA_SRC_BLEND_INV_SRCCOLOR, D3DBLEND_INVSRCCOLOR,
CHAPTER 14. THE FRAME BUFFER

IDM_ALPHA_DEST_BLEND_DEST_ALPHA, D3DBLEND_DESTALPHA,
IDM_ALPHA_DEST_BLEND_INV_DEST_ALPHA, D3DBLEND_INVDESTALPHA,
IDM_ALPHA_DEST_BLEND_DEST_COLOR, D3DBLEND_DESTCOLOR,
IDM_ALPHA_DEST_BLEND_INV_DEST_COLOR, D3DBLEND_INVDESTCOLOR,
IDM_ALPHA_DEST_BLEND_SRC_ALPHA_SAT, D3DBLEND_SRCALPHASAT,
IDM_ALPHA_DEST_BLEND_FACTOR, D3DBLEND_BLENDFACTOR,
IDM_ALPHA_DEST_BLEND_INV_FACTOR, D3DBLEND_INVBLENDFACTOR

HMENU menu = TWS(::GetMenu(m_hWnd));
for (UINT i = 0; i < NUM_OF(data); i++)
{
    MENUITEMINFO info = {
        sizeof(info), MIIM_DATA
    };
    info.dwItemData = data[i].m_data;
    TWS(::SetMenuItemInfo(menu, data[i].m_id, FALSE, &info));
}

DWORD CMYD3DApplication::get_menu_data(DWORD id)
{
    MENUITEMINFO info = {
        sizeof(info), MIIM_DATA
    };
    THR(::GetMenuItemInfo(TWS(::GetMenu(m_hWnd)), id, FALSE, &info));
    return DWORD(info.dwItemData);
}

LRESULT CMYD3DApplication::on_left_button_down(HWND window, WPARAM, LPARAM lp, bool &handled)
{
    ::SetCapture(window);
    m_scissor_rect.left = GET_X_LPARAM(lp);
    m_scissor_rect.top = GET_Y_LPARAM(lp);
    m_scissor_rect.right = GET_X_LPARAM(lp);
    m_scissor_rect.bottom = GET_Y_LPARAM(lp);
    THR(m_pd3dDevice->SetScissorRect(&m_scissor_rect));
    handled = true;
    return LRESULT(0);
}

LRESULT CMYD3DApplication::on_mouse_move(HWND window, WPARAM, LPARAM lp, bool &handled)
{
    if (window == ::GetCapture())
    
    

m_scissor_rect.right = std::min(m_d3dsdBackBuffer.Width, UINT(GET_X_LPARAM(lp)));  

m_scissor_rect.bottom = std::min(m_d3dsdBackBuffer.Height, UINT(GET_Y_LPARAM(lp)));  

RECT tmp = m_scissor_rect;  

if (tmp.right < tmp.left)  
{
    std::swap(tmp.right, tmp.left);  
}

if (tmp.bottom < tmp.top)  
{
    std::swap(tmp.bottom, tmp.top);  
}

THR(m_pd3dDevice->SetScissorRect(&tmp));  

handled = true;  

return LRESULT(0);  

return LRESULT(0);  

CMyD3DApplication::on_left_button_up(HWND window, WPARAM, LPARAM, bool &handled)  
{
    if (window == ::GetCapture())  
    {
        TWS(::ReleaseCapture());  
        handled = true;  
    }
    return LRESULT(0);  
}