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 ancilliary 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 vlaue, 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 visibilty 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

$$\alpha_s \ \langle op \rangle \ \alpha_r$$

where α_s is the alpha value of the source pixel, α_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 D3DPCMPCAPS_NEVER        0x00000001L
#define D3DPCMPCAPS_LESS         0x00000002L
#define D3DPCMPCAPS_EQUAL        0x00000004L
#define D3DPCMPCAPS_LESSEQUAL    0x00000008L
#define D3DPCMPCAPS_GREATER      0x00000010L
#define D3DPCMPCAPS_NOTEQUAL     0x00000020L
#define D3DPCMPCAPS_GREATEREQUAL 0x00000040L
#define D3DPCMPCAPS_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 targes. 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.

```
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 taget. 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.

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

```
    D3DZB_USEW  = 2,
} D3DZBUFFERTYPE;
```

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

$$z_s \ \langle op \rangle \ z_d$$

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 $\langle op \rangle$ is the comparison operator used. RS Z Func defines the comparison function $\langle op \rangle$ and is given by a member of the D3DCMPFUNC enumeration.

When the D3DPRASTERCAPS_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 D3DPCMPCAPS 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.

```
#define D3DPRASTERCAPS_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 D3DPRASTERCAPS_WBUFFER bit of D3DCAPS9::RasterCaps is set. A device supports W based fog if the D3DPRASTERCAPS_WFOG bit of D3DCAPS::RasterCaps is set. The MaxVertexW member of D3DCAPS9 gives the maximum W value for a vertex supported by the device.

```
#define D3DPRASTERCAPS_WBUFFER 0x00040000L
#define D3DPRASTERCAPS_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

TODO: Fix bias description

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 DDDPRasterCapsZBias 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[1].

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

```
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 D3DSTENCILCAPS flags.

[1] Calling Clear with D3DCLEAR_STENCIL when no stencil buffer is present is an error.

Enumerant	Operation
D3DSTENCILOP_KEEP	Keeps the existing destination stencil buffer value as the source stencil value.
D3DSTENCILOP_ZERO	Uses a value of zero as the source stencil value.
D3DSTENCILOP_REPLACE	Uses the stencil reference value as the source stencil value.
D3DSTENCILOP_INVERT	Inverts the destination stencil buffer value as the source stencil value. That is, the source value is the bitwise complement of the destination value.
D3DSTENCILOP_INCR	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.
D3DSTENCILOP_DECR	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.
D3DSTENCILOP_INCRSAT	As D3DSTENCILOP_INCR, but clamp to the maximum stencil value 2^s.
D3DSTENCILOP_DECRSAT	As D3DSTENCILOP_DECR, but clamp to zero.

Table 14.1: Summary of the stencil operations corresponding to D3DSTENCILOP.

```
#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
```

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.

```
RS Stencil Enable = FALSE
RS Stencil Func   = D3DCMP_ALWAYS
RS Stencil Ref    = 1
RS Stencil Fail   = D3DSTENCILOP_REPLACE
RS Stencil Z Fail = D3DSTENCILOP_REPLACE
RS Stencil Pass   = D3DSTENCILOP_REPLACE
```

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.

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 alpha 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 whereby 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:

$$\begin{aligned} C &= \langle r, g, b, a \rangle \\ f &= \langle f_r, f_g, f_b, f_a \rangle \\ Cf &= \langle rf_r, gf_g, bf_b, af_a \rangle \\ C'_s &= \langle func \rangle (C_s f_s, C_d f_d) \end{aligned}$$

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, $\langle func \rangle$ is the blend function and C'_s 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.

RS Src Blend and 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 $\langle func \rangle$ is specified by 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.

```
typedef enum _D3DBLEND {
    D3DBLEND_ZERO            = 1,
    D3DBLEND_ONE             = 2,
    D3DBLEND_SRCCOLOR        = 3,
    D3DBLEND_INVSRCCOLOR     = 4,
    D3DBLEND_SRCALPHA        = 5,
    D3DBLEND_INVSRCALPHA     = 6,
    D3DBLEND_DESTALPHA       = 7,
    D3DBLEND_INVDESTALPHA    = 8,
    D3DBLEND_DESTCOLOR       = 9,
    D3DBLEND_INVDESTCOLOR    = 10,
    D3DBLEND_SRCALPHASAT     = 11,
    D3DBLEND_BOTHINVSRCALPHA = 13
} D3DBLEND;

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 D3DSWAPEFFECT_FLIP or D3DSWAPEFFECT_DISCARD, a device will properly support destination alpha if the D3DCAPS3_ALPHA_FULLSCREEN_FLIP_OR_DISCARD bit of D3DCAPS9::Caps3 is set. Otherwise, an application should use D3DSWAPEFFECT_COPY or DDDSwapEffectCopyVSync

TODO: FIX VSYNC

Enumerant	Blend Factor f
D3DBLEND_ZERO	$\langle 0, 0, 0, 0\rangle$
D3DBLEND_ONE	$\langle 1, 1, 1, 1\rangle$
D3DBLEND_SRCCOLOR	$\langle r_s, g_s, b_s, a_s\rangle$
D3DBLEND_INVSRCCOLOR	$\langle 1 - r_s, 1 - g_s, 1 - b_s, 1 - a_s\rangle$
D3DBLEND_SRCALPHA	$\langle a_s, a_s, a_s, a_s\rangle$
D3DBLEND_INVSRCALPHA	$\langle 1 - a_s, 1 - a_s, 1 - a_s, 1 - a_s\rangle$
D3DBLEND_DESTALPHA	$\langle a_d, a_d, a_d, a_d\rangle$
D3DBLEND_INVDESTALPHA	$\langle 1 - a_d, 1 - a_d, 1 - a_d, 1 - a_d\rangle$
D3DBLEND_DESTCOLOR	$\langle r_d, g_d, b_d, a_d\rangle$
D3DBLEND_INVDESTCOLOR	$\langle 1 - r_d, 1 - g_d, 1 - b_d, 1 - a_d\rangle$
D3DBLEND_SRCALPHASAT	$\langle f, f, f, f\rangle,\ f = \min(a_s, 1 - a_d)$
D3DBLEND_BOTHINVSRCALPHA[1]	$\langle 1 - a_s, 1 - a_s, 1 - a_s, 1 - a_s\rangle$ $\langle a_s, a_s, a_s, a_s\rangle$

Table 14.2: Blend factors corresponding to the enumerants of D3DBLEND. The source pixel color is $\langle r_s, g_s, b_s, a_s\rangle$ and the destination pixel color is $\langle r_d, g_d, b_d, a_d\rangle$. [1]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.

Enumerant	Blend Function
D3DBLENDOP_ADD	$\langle func\rangle(s, d) = s + d$
D3DBLENDOP_SUBTRACT	$\langle func\rangle(s, d) = s - d$
D3DBLENDOP_REVSUBTRACT	$\langle func\rangle(s, d) = d - s$
D3DBLENDOP_MIN	$\langle func\rangle(s, d) = \min(s, d)$
D3DBLENDOP_MAX	$\langle func\rangle(s, d) = \max(s, d)$

Table 14.3: Blending functions corresponding to the enumerants of D3DBLENDOP. 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.

Operation f_s f_d	Diagram	Operation f_s f_d	Diagram
S $1 \quad 0$		D $0 \quad 1$	
clear $0 \quad 0$		S **xor** D $1-\alpha_d \quad 1-\alpha_s$	
S **over** D $1 \quad 1-\alpha_s$		D **over** S $1-\alpha_d \quad 1$	
S **in** D $\alpha_d \quad 0$		D **in** S $0 \quad \alpha_s$	
S **out** D $1-\alpha_d \quad 0$		D **out** S $0 \quad 1-\alpha_s$	
S **atop** D $\alpha_d \quad 1-\alpha_s$		D **atop** S $1-\alpha_d \quad \alpha_s$	

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

if destination alpha is required in exclusive mode. When the D3DPMISCCAPS_-BLENDOP bit of D3DCAPS9::PrimitiveMiscCaps is set, the device supports the extended blend operators in the D3DBLENDOP enumeration. D3DBLENDOP_ADD is always supported.

```
#define D3DCAPS3_ALPHA_FULLSCREEN_FLIP_OR_DISCARD 0x00000020L
#define D3DPMISCCAPS_BLENDOP                      0x00000800L
```

The SrcBlendCaps and DestBlendCaps members of D3DCAPS9 define the supported source and destination blend factors, respectively. Each bit that is set corresponds to a different blend factor. The bits are given by the D3DP-BLENDCAPS flags. The D3DPBLENDCAPS_BOTHINVSRCALPHA bit is only valid for SrcBlendCaps.

```
#define D3DPBLENDCAPS_ZERO            0x00000001L
#define D3DPBLENDCAPS_ONE             0x00000002L
#define D3DPBLENDCAPS_SRCCOLOR        0x00000004L
#define D3DPBLENDCAPS_INVSRCCOLOR     0x00000008L
#define D3DPBLENDCAPS_SRCALPHA        0x00000010L
#define D3DPBLENDCAPS_INVSRCALPHA     0x00000020L
#define D3DPBLENDCAPS_DESTALPHA       0x00000040L
#define D3DPBLENDCAPS_INVDESTALPHA    0x00000080L
#define D3DPBLENDCAPS_DESTCOLOR       0x00000100L
#define D3DPBLENDCAPS_INVDESTCOLOR    0x00000200L
#define D3DPBLENDCAPS_SRCALPHASAT     0x00000400L
#define D3DPBLENDCAPS_BOTHINVSRCALPHA 0x00001000L
```

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.

TODO: Raster Caps Stretch Blt Multi Sample

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.

```
#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,

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 D3DCOLORWRITEENABLE flag bits. Each set bit corresponds to a channel that is enabled for writing.

```
#define D3DCOLORWRITEENABLE_RED   (1L<<0)
#define D3DCOLORWRITEENABLE_GREEN (1L<<1)
#define D3DCOLORWRITEENABLE_BLUE  (1L<<2)
#define D3DCOLORWRITEENABLE_ALPHA (1L<<3)
```

The color write mask is supported if the D3DPMISCCAPS_COLORWRITEENABLE bit of D3DCAPS9::PrimitiveMiscCaps is set.

```
#define D3DPMISCCAPS_COLORWRITEENABLE 0x00000080L
```

Using alpha blending with RS Src Blend and RS Dest Blend set to D3DBLEND_ZERO and 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 ith bit of the mask, where the least significant bit is numbered 0, enables writing to the ith sample in the render target.

14.10 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.

Multsampling 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

`-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.

```
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
11 #define STRICT
12 #include <windows.h>
13 #include <windowsx.h>
14 #include <commctrl.h>
15
16 #include <atlbase.h>
17
18 #include <d3dx9.h>
19
20 #include "DXUtil.h"
21 #include "D3DEnumeration.h"
22 #include "D3DSettings.h"
23 #include "D3DApp.h"
24 #include "D3DFont.h"
25 #include "D3DUtil.h"
26
27 #include "rt/app.h"
28 #include "rt/hr.h"
29 #include "rt/mat.h"
30 #include "rt/mesh.h"
31 #include "rt/misc.h"
32 #include "rt/rt_ColorSel.h"
33 #include "rt/states.h"
34 #include "rt/surface.h"
```

```
35  #include "rt/texture.h"
36  #include "rt/tstring.h"
37  #include "rt/vertexbuf.h"
38
39  #include "resource.h"
40  #include "rt_FrameBuffer.h"
41
42  const UINT
43  CMyD3DApplication::STIPPLE_SIZE = 64;
44  const UINT
45  CMyD3DApplication::TILE_SIZE = 64;
46
47  //////////////////////////////////////////////////////////////////////
48  // CMyD3DApplication::s_screen_vertex::FVF
49  //
50  // FVF code for screen-space vertices used to draw the stipple texture
51  //
52  const DWORD CMyD3DApplication::s_screen_vertex::FVF =
53      D3DFVF_XYZRHW | D3DFVF_TEX1 | D3DFVF_TEXCOORDSIZE2(0);
54
55  //////////////////////////////////////////////////////////////////////
56  // g_jitterN, sm_jitter
57  //
58  // jitter coefficients used for depth of field multisampling
59  //
60  const float g_jitter1[2] =
61  {
62      0, 0
63  };
64  const float g_jitter2[4] =
65  {
66      0.25f,          0.75f,
67      0.75f,          0.25f
68  };
69  const float g_jitter3[6] =
70  {
71      0.5033922635f,  0.8317967229f,
72      0.7806016275f,  0.2504380877f,
73      0.2261828938f,  0.4131553612f
74  };
75  const float g_jitter4[8] =
76  {
77      0.375f,         0.25f,
78      0.125f,         0.75f,
79      0.875f,         0.25f,
80      0.625f,         0.75f
```

```
81  };
82  const float g_jitter5[10] =
83  {
84      0.5f,           0.5f,
85      0.3f,           0.1f,
86      0.7f,           0.9f,
87      0.9f,           0.3f,
88      0.1f,           0.7f
89  };
90  const float g_jitter6[12] =
91  {
92      0.4646464646f,  0.4646464646f,
93      0.1313131313f,  0.7979797979f,
94      0.5353535353f,  0.8686868686f,
95      0.8686868686f,  0.5353535353f,
96      0.7979797979f,  0.1313131313f,
97      0.2020202020f,  0.2020202020f
98  };
99  const float g_jitter8[16] =
100 {
101     0.5625f,        0.4375f,
102     0.0625f,        0.9375f,
103     0.3125f,        0.6875f,
104     0.6875f,        0.8125f,
105     0.8125f,        0.1875f,
106     0.9375f,        0.5625f,
107     0.4375f,        0.0625f,
108     0.1875f,        0.3125f
109 };
110 const float g_jitter9[18] =
111 {
112     0.5f,           0.5f,
113     0.1666666666f,  0.944444444f,
114     0.5f,           0.1666666666f,
115     0.5f,           0.8333333333f,
116     0.1666666666f,  0.2777777777f,
117     0.8333333333f,  0.3888888888f,
118     0.1666666666f,  0.6111111111f,
119     0.8333333333f,  0.7222222222f,
120     0.8333333333f,  0.0555555555f,
121 };
122 const float g_jitter12[24] =
123 {
124     0.4166666666f,  0.625f,
125     0.9166666666f,  0.875f,
126     0.25f,          0.375f,
```

```
127         0.4166666666f,  0.125f,
128         0.75f,          0.125f,
129         0.0833333333f,  0.125f,
130         0.75f,          0.625f,
131         0.25f,          0.875f,
132         0.5833333333f,  0.375f,
133         0.9166666666f,  0.375f,
134         0.0833333333f,  0.625f,
135         0.5833333333f,  0.875f
136     };
137     const float g_jitter16[32] =
138     {
139         0.375f,         0.4375f,
140         0.625f,         0.0625f,
141         0.875f,         0.1875f,
142         0.125f,         0.0625f,
143         0.375f,         0.6875f,
144         0.875f,         0.4375f,
145         0.625f,         0.5625f,
146         0.375f,         0.9375f,
147         0.625f,         0.3125f,
148         0.125f,         0.5625f,
149         0.125f,         0.8125f,
150         0.375f,         0.1875f,
151         0.875f,         0.9375f,
152         0.875f,         0.6875f,
153         0.125f,         0.3125f,
154         0.625f,         0.8125f
155     };
156     const float *
157     CMyD3DApplication::sm_jitter[16] =
158     {
159         g_jitter1,
160         g_jitter2,
161         g_jitter3,
162         g_jitter4,
163         g_jitter5,
164         g_jitter6,
165         g_jitter8,
166         g_jitter8,
167         g_jitter9,
168         g_jitter12,
169         g_jitter12,
170         g_jitter12,
171         g_jitter16,
172         g_jitter16,
```

```
173     g_jitter16,
174     g_jitter16
175 };
176 
177 ////////////////////////////////////////////////////////
178 // determine blend factors based on blending operator
179 const struct CMyD3DApplication::s_blend_factor
180 CMyD3DApplication::sm_blend_factors[12] =
181 {
182     // in order of e_blending_operator:
183     // E_COMPOSITE_CLEAR
184     D3DBLEND_ZERO,          D3DBLEND_ZERO,
185     // E_COMPOSITE_SRC
186     D3DBLEND_ONE,           D3DBLEND_ZERO,
187     // E_COMPOSITE_DEST
188     D3DBLEND_ZERO,          D3DBLEND_ONE,
189     // E_COMPOSITE_SRC_OVER_DEST
190     D3DBLEND_ONE,           D3DBLEND_INVSRCALPHA,
191     // E_COMPOSITE_DEST_OVER_SRC
192     D3DBLEND_INVDESTALPHA,  D3DBLEND_ONE,
193     // E_COMPOSITE_SRC_IN_DEST
194     D3DBLEND_DESTALPHA,     D3DBLEND_ZERO,
195     // E_COMPOSITE_DEST_IN_SRC
196     D3DBLEND_ZERO,          D3DBLEND_SRCALPHA,
197     // E_COMPOSITE_SRC_ATOP_DEST
198     D3DBLEND_DESTALPHA,     D3DBLEND_INVSRCALPHA,
199     // E_COMPOSITE_DEST_ATOP_SRC
200     D3DBLEND_INVDESTALPHA,  D3DBLEND_SRCALPHA,
201     // E_COMPOSITE_SRC_OUT_DEST
202     D3DBLEND_INVDESTALPHA,  D3DBLEND_ZERO,
203     // E_COMPOSITE_DEST_OUT_SRC
204     D3DBLEND_ZERO,          D3DBLEND_INVSRCALPHA,
205     // E_COMPOSITE_SRC_XOR_DEST
206     D3DBLEND_INVDESTALPHA,  D3DBLEND_INVSRCALPHA
207 };
208 
209 //-----------------------------------------------------------------------------
210 // Global access to the app (needed for the global WndProc())
211 //-----------------------------------------------------------------------------
212 CMyD3DApplication* g_pApp  = NULL;
213 HINSTANCE          g_hInst = NULL;
214 
215 
216 
217 
218 ////////////////////////////////////////////////////////////////////////
```

```
219 // parse_command_line
220 //
221 // Parse the command line, looking for the switches:
222 //
223 //  -nodest     Don't use destination alpha
224 //  -nostencil  Don't use stencil planes
225 //
226 // Throw up a message box if we encounter something we don't recognize.
227 //
228 bool
229 parse_command_line(LPCTSTR command_line, DWORD &stencil_bits)
230 {
231     rt::tistringstream args(command_line);
232     rt::tstring arg;
233     while (!args.eof())
234     {
235         args >> arg;
236         if (_T("-nostencil") == arg)
237         {
238             stencil_bits = 0;
239         }
240         else if (arg.length())
241         {
242             rt::tostringstream barf;
243             barf << _T("I did not recognize the switch '") << arg
244                 << _T("' in the command line\n\"") << command_line
245                 << _T("\".\n\nValid switches are:\n")
246                 << _T("-nostencil\t\tDon't use stencil planes.\n");
247             ::MessageBox(NULL, barf.str().c_str(),
248                 _T("rt_FrameBuffer: Invalid Command Line Argument"), MB_OK);
249 
250             return false;
251         }
252     }
253 
254     return true;
255 }
256 
257 //-----------------------------------------------------------------------------
258 // Name: _tWinMain()
259 // Desc: Entry point to the program. Initializes everything, and goes into a
260 //       message-processing loop. Idle time is used to render the scene.
261 //-----------------------------------------------------------------------------
262 INT WINAPI
263 _tWinMain(HINSTANCE instance, HINSTANCE, LPTSTR command_line, int)
264 {
```

```
265       DWORD stencil_bits = 2;
266       if (!parse_command_line(command_line, stencil_bits))
267       {
268           return 0;
269       }
270
271       try
272       {
273           CMyD3DApplication d3dApp(stencil_bits);
274
275           g_pApp  = &d3dApp;
276           g_hInst = instance;
277
278           ::InitCommonControls();
279           if (FAILED(d3dApp.Create(instance)))
280               return 0;
281
282           return d3dApp.Run();
283       }
284       catch (rt::hr_message &bang)
285       {
286           return rt::display_error(bang);
287       }
288       catch (...)
289       {
290           return E_UNEXPECTED;
291       }
292   }
293
294   HMENU
295   find_menu(HMENU menu, const rt::tstring &text)
296   {
297       UINT count = ::GetMenuItemCount(menu);
298       for (UINT i = 0; i < count; i++)
299       {
300           MENUITEMINFO info = { sizeof(info), MIIM_TYPE | MIIM_SUBMENU };
301           TWS(::GetMenuItemInfo(menu, i, TRUE, &info));
302           if ((MFT_STRING == info.fType) && info.hSubMenu)
303           {
304               std::vector<TCHAR> buff(info.cch+2);
305               info.dwTypeData = &buff[0];
306               info.cch = buff.size();
307               TWS(::GetMenuItemInfo(menu, i, TRUE, &info));
308               if (text == info.dwTypeData)
309               {
310                   return info.hSubMenu;
```

```
311             }
312         }
313     }
314
315     return HMENU(NULL);
316 }
317
318 void
319 enable_menu(HMENU menu, const rt::tstring &text, bool enabled)
320 {
321     UINT count = ::GetMenuItemCount(menu);
322     for (UINT i = 0; i < count; i++)
323     {
324         MENUITEMINFO info =
325         { sizeof(info), MIIM_STATE | MIIM_TYPE | MIIM_SUBMENU };
326         TWS(::GetMenuItemInfo(menu, i, TRUE, &info));
327         if ((MFT_STRING == info.fType) && info.hSubMenu)
328         {
329             std::vector<TCHAR> buff(info.cch+2);
330             info.dwTypeData = &buff[0];
331             info.cch = buff.size();
332             TWS(::GetMenuItemInfo(menu, i, TRUE, &info));
333             if (text == info.dwTypeData)
334             {
335                 info.fState &= ~(MFS_ENABLED | MFS_DISABLED);
336                 info.fState |= enabled ? MFS_ENABLED : MFS_DISABLED;
337                 info.fMask = MIIM_STATE;
338                 TWS(::SetMenuItemInfo(menu, i, TRUE, &info));
339                 return;
340             }
341         }
342     }
343     // should have found it
344     ATLASSERT(false);
345 }
346
347 //-----------------------------------------------------------------------------
348 // Name: CMyD3DApplication()
349 // Desc: Application constructor.   Paired with ~CMyD3DApplication()
350 //       Member variables should be initialized to a known state here.
351 //       The application window has not yet been created and no Direct3D device
352 //       has been created, so any initialization that depends on a window or
353 //       Direct3D should be deferred to a later stage.
354 //-----------------------------------------------------------------------------
355 CMyD3DApplication::CMyD3DApplication(DWORD stencil_bits) :
356     CD3DApplication(),
```

```
357       m_teapot(),
358       m_texture(),
359       m_tile(),
360       m_device_tile(),
361       m_torus(),
362       m_stipple_verts(),
363       m_stipple(),
364       m_tile_rects(),
365       m_tile_offsets(),
366       m_texture_file(_T("dx5_logo.bmp")),
367       m_num_stipple_quads(0),
368       m_text_fg(D3DCOLOR_XRGB(255, 255, 0)),
369       m_specular(D3DCOLOR_XRGB(255, 255, 255)),
370       m_bg(D3DCOLOR_XRGB(0, 0, 0)),
371       m_fg(D3DCOLOR_XRGB(255, 255, 255)),
372       m_light_color(D3DCOLOR_XRGB(255, 255, 255)),
373       m_blend_factor(D3DCOLOR_ARGB(128, 128, 128, 128)),
374       m_color_composite(E_COMPOSITE_NONE),
375       m_alpha_composite(E_COMPOSITE_NONE),
376       m_color_blend_op(D3DBLENDOP_ADD),
377       m_color_src_blend(D3DBLEND_ONE),
378       m_color_dest_blend(D3DBLEND_ZERO),
379       m_alpha_blend_op(D3DBLENDOP_ADD),
380       m_alpha_src_blend(D3DBLEND_ONE),
381       m_alpha_dest_blend(D3DBLEND_ZERO),
382       m_can_separate_alpha_blend(false),
383       m_color_allow_dest_alpha(false),
384       m_color_blend_enable(true),
385       m_alpha_allow_dest_alpha(false),
386       m_alpha_blend_enable(false),
387       m_multisample_motion_blur(false),
388       m_multisample_depth_of_field(false),
389       m_multisample_antialias(false),
390       m_color_write_enable_red(true),
391       m_color_write_enable_green(true),
392       m_color_write_enable_blue(true),
393       m_color_write_enable_alpha(true),
394       m_z_write_enable(true),
395       m_textured(false),
396       m_stencil_stippling(false),
397       m_stencil_mask(false),
398       m_show_stats(true),
399       m_animate_view(false),
400       m_dithered(false),
401       m_can_texture(false),
402       m_destination_alpha(false),
```

```
403       m_tile_background(false),
404       m_build_stencil(false),
405       m_can_scissor(false),
406       m_scissor_enable(false),
407       m_last_rot_x(0.0f),
408       m_last_rot_y(0.0f),
409       m_bLoadingApp(TRUE),
410       m_font(_T("Arial"), 12, D3DFONT_BOLD),
411       m_input(),
412       m_rot_x(0.0f),
413       m_rot_y(0.0f)
414   {
415       m_dwCreationWidth           = 500;
416       m_dwCreationHeight          = 375;
417       m_strWindowTitle            = _T("rt_FrameBuffer");
418       m_d3dEnumeration.AppUsesDepthBuffer   = TRUE;
419           m_bStartFullscreen                      = false;
420           m_bShowCursorWhenFullscreen     = false;
421
422       RECT zero = { 0 };
423       m_scissor_rect = zero;
424
425       // Read settings from registry
426       ReadSettings();
427
428       m_d3dEnumeration.AppMinStencilBits = stencil_bits;
429   }
430
431
432
433
434   //--------------------------------------------------------------------------
435   // Name: ~CMyD3DApplication()
436   // Desc: Application destructor.  Paired with CMyD3DApplication()
437   //--------------------------------------------------------------------------
438   CMyD3DApplication::~CMyD3DApplication()
439   {
440   }
441
442
443
444
445   //--------------------------------------------------------------------------
446   // Name: OneTimeSceneInit()
447   // Desc: Paired with FinalCleanup().
448   //       The window has been created and the IDirect3D9 interface has been
```

```
449 //       created, but the device has not been created yet.  Here you can
450 //       perform application-related initialization and cleanup that does
451 //       not depend on a device.
452 //-----------------------------------------------------------------------------
453 HRESULT CMyD3DApplication::OneTimeSceneInit()
454 {
455     // Drawing loading status message until app finishes loading
456     ::SendMessage(m_hWnd, WM_PAINT, 0, 0);
457
458     m_bLoadingApp = FALSE;
459
460     set_menu_data();
461
462     return S_OK;
463 }
464
465
466
467
468 //-----------------------------------------------------------------------------
469 // Name: ReadSettings()
470 // Desc: Read the app settings from the registry
471 //-----------------------------------------------------------------------------
472 VOID CMyD3DApplication::ReadSettings()
473 {
474     HKEY hkey;
475     if (ERROR_SUCCESS == ::RegCreateKeyEx(HKEY_CURRENT_USER, DXAPP_KEY,
476         0, NULL, REG_OPTION_NON_VOLATILE, KEY_ALL_ACCESS, NULL, &hkey, NULL))
477     {
478         // Read the stored window width/height.  This is just an example,
479         // of how to use ::DXUtil_Read*() functions.
480         ::DXUtil_ReadIntRegKey(hkey, _T("Width"), &m_dwCreationWidth, m_dwCreati
481         ::DXUtil_ReadIntRegKey(hkey, _T("Height"), &m_dwCreationHeight, m_dwCrea
482
483         ::RegCloseKey(hkey);
484     }
485 }
486
487
488
489
490 //-----------------------------------------------------------------------------
491 // Name: WriteSettings()
492 // Desc: Write the app settings to the registry
493 //-----------------------------------------------------------------------------
494 VOID CMyD3DApplication::WriteSettings()
```

```
495 {
496     HKEY hkey;
497 
498     if (ERROR_SUCCESS == ::RegCreateKeyEx(HKEY_CURRENT_USER, DXAPP_KEY,
499         0, NULL, REG_OPTION_NON_VOLATILE, KEY_ALL_ACCESS, NULL, &hkey, NULL))
500     {
501         // Write the window width/height.  This is just an example,
502         // of how to use ::DXUtil_Write*() functions.
503         ::DXUtil_WriteIntRegKey(hkey, _T("Width"), m_rcWindowClient.right);
504         ::DXUtil_WriteIntRegKey(hkey, _T("Height"), m_rcWindowClient.bottom);
505 
506         ::RegCloseKey(hkey);
507     }
508 }
509 
510 //--------------------------------------------------------------------------
511 // Name: InitDeviceObjects()
512 // Desc: Paired with DeleteDeviceObjects()
513 //       The device has been created.  Resources that are not lost on
514 //       Reset() can be created here -- resources in D3DPOOL_MANAGED,
515 //       D3DPOOL_SCRATCH, or D3DPOOL_SYSTEMMEM.  Image surfaces created via
516 //       CreateImageSurface are never lost and can be created here.  Vertex
517 //       shaders and pixel shaders can also be created here as they are not
518 //       lost on Reset().
519 //--------------------------------------------------------------------------
520 HRESULT CMyD3DApplication::InitDeviceObjects()
521 {
522     // rather than reject devices without this cap, we simply disable
523     // texturing and allow rendering with the diffuse color on this device.
524     m_can_texture =
525              (m_d3dCaps.TextureFilterCaps & D3DPTFILTERCAPS_MINFLINEAR) &&
526              (m_d3dCaps.TextureFilterCaps & D3DPTFILTERCAPS_MAGFLINEAR) &&
527              (m_d3dCaps.TextureFilterCaps & D3DPTFILTERCAPS_MIPFLINEAR) &&
528              (m_d3dCaps.TextureOpCaps & D3DTEXOPCAPS_SELECTARG2) &&
529              (m_d3dCaps.TextureOpCaps & D3DTEXOPCAPS_MODULATE) &&
530              (m_d3dCaps.TextureAddressCaps & D3DPTADDRESSCAPS_WRAP) &&
531              (m_d3dCaps.VertexProcessingCaps & D3DVTXPCAPS_TEXGEN);
532     if (m_can_texture)
533     {
534         create_texture();
535         {
536             DWORD passes = 0;
537             const HRESULT hr = m_pd3dDevice->ValidateDevice(&passes);
538             if (FAILED(hr))
539             {
540                 m_can_texture = false;
```

```
541             }
542         }
543     }
544     if (!m_can_texture)
545     {
546         m_textured = false;
547         m_texture = 0;
548     }
549
550     // Create a teapot mesh using D3DX and optimize it
551     {
552         rt::dx_buffer<DWORD> adj;
553         THR(::D3DXCreateTeapot(m_pd3dDevice, &m_teapot, &adj));
554         THR(m_teapot->OptimizeInplace(D3DXMESHOPT_ATTRSORT, adj,
555             NULL, NULL, NULL));
556     }
557
558     // create a background ARGB tile with three steps in alpha in Y
559     // and a gradient in rgb in X
560     THR(m_pd3dDevice->CreateOffscreenPlainSurface(TILE_SIZE, TILE_SIZE,
561         D3DFMT_A8R8G8B8, D3DPOOL_SCRATCH, &m_tile, NULL));
562     {
563         rt::surface_lock lock(m_tile);
564         for (UINT y = 0; y < TILE_SIZE; y++)
565         {
566             DWORD *row = lock.scanline32(y);
567             BYTE alpha = 0;
568             if (y < TILE_SIZE/3)
569             {
570                 alpha = 255/3;
571             }
572             else if (y < 2*TILE_SIZE/3)
573             {
574                 alpha = 255*2/3;
575             }
576             else
577             {
578                 alpha = 255;
579             }
580             for (UINT x = 0; x < TILE_SIZE; x++)
581             {
582                 const BYTE rgb = BYTE(alpha*x/(TILE_SIZE-1));
583                 row[x] = D3DCOLOR_ARGB(alpha, rgb, rgb, rgb);
584             }
585         }
586     }
```

```
587
588       // create a torus used to create the stencil mask
589       THR(::D3DXCreateTorus(m_pd3dDevice, 0.325f, 0.65f, 10, 50, &m_torus, NULL));
590
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   //-----------------------------------------------------------------------------
623   // Name: RestoreDeviceObjects()
624   // Desc: Paired with InvalidateDeviceObjects()
625   //       The device exists, but may have just been Reset().  Resources in
626   //       D3DPOOL_DEFAULT and any other device state that persists during
627   //       rendering should be set here.  Render states, matrices, textures,
628   //       etc., that don't change during rendering can be set once here to
629   //       avoid redundant state setting during Render() or FrameMove().
630   //-----------------------------------------------------------------------------
631   HRESULT CMyD3DApplication::RestoreDeviceObjects()
632   {
```

```
633     HMENU menu = TWS(::GetMenu(m_hWnd));
634
635     m_can_scissor = (m_d3dCaps.RasterCaps & D3DPRASTERCAPS_SCISSORTEST) != 0;
636     rt::enable_menu(menu, ID_OPTIONS_SCISSORTEST, m_can_scissor);
637     if (m_scissor_enable)
638     {
639         m_scissor_enable = false;
640         rt::check_menu(menu, ID_OPTIONS_SCISSORTEST, false);
641     }
642
643     update_blending(menu, false);
644     m_destination_alpha = (D3DFMT_A8R8G8B8 == m_d3dsdBackBuffer.Format) ||
645         (D3DFMT_A1R5G5B5 == m_d3dsdBackBuffer.Format) ||
646         (D3DFMT_A2R10G10B10 == m_d3dsdBackBuffer.Format);
647     m_can_separate_alpha_blend =
648         0 != (m_d3dCaps.PrimitiveMiscCaps & D3DPMISCCAPS_SEPARATEALPHABLEND);
649     update_blending(menu, true);
650
651     // if D3DRS_COLORWRITEENABLE is not supported, disable menu items
652     {
653         const bool mask_colors = 0 !=
654             (m_d3dCaps.PrimitiveMiscCaps & D3DPMISCCAPS_COLORWRITEENABLE);
655         rt::enable_menu(menu, IDM_COLOR_WRITE_ENABLE_RED, mask_colors);
656         rt::enable_menu(menu, IDM_COLOR_WRITE_ENABLE_GREEN, mask_colors);
657         rt::enable_menu(menu, IDM_COLOR_WRITE_ENABLE_BLUE, mask_colors);
658         rt::enable_menu(menu, IDM_COLOR_WRITE_ENABLE_ALPHA, mask_colors);
659     }
660
661     // can't do any stencil effects without stencil bits
662     {
663         const bool support = (m_d3dEnumeration.AppMinStencilBits >= 2);
664         if (!support)
665         {
666             m_stencil_mask = false;
667             m_stencil_stippling = false;
668             rt::check_menu(menu, IDM_STENCIL_IRREGULAR_MASK, false);
669             rt::check_menu(menu, IDM_STENCIL_STIPPLING, false);
670         }
671         enable_menu(menu, _T("&Stenciling"), support);
672     }
673
674     // can't do separate alpha channel blend
675     enable_menu(menu, _T("&Alpha Blend"), m_can_separate_alpha_blend);
676
677     // can't do any of these without multisampling
678     if (D3DMULTISAMPLE_NONE == m_d3dpp.MultiSampleType)
```

```
679         {
680             m_multisample_antialias = false;
681             m_multisample_depth_of_field = false;
682             m_multisample_motion_blur = false;
683 
684             rt::check_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, false);
685             rt::check_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, false);
686             rt::check_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, false);
687 
688             enable_menu(menu, _T("&Multisampling"), false);
689             //rt::enable_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, false);
690             //rt::enable_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, false);
691             //rt::enable_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, false);
692         }
693     #if 0
694         // if this cap is set, can't do motion blur or depth of field
695         else if (m_d3dCaps.RasterCaps & D3DPRASTERCAPS_STRETCHBLTMULTISAMPLE)
696         {
697             m_multisample_depth_of_field = false;
698             m_multisample_motion_blur = false;
699 
700             rt::check_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, false);
701             rt::check_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, false);
702 
703             rt::enable_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, true);
704             rt::enable_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, false);
705             rt::enable_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, false);
706         }
707     #endif
708         else
709         {
710             enable_menu(menu, _T("&Multisampling"), true);
711             //rt::enable_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, true);
712             //rt::enable_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, true);
713             //rt::enable_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, true);
714         }
715         TWS(::DrawMenuBar(m_hWnd));
716 
717         // Set up our view matrix. A view matrix can be defined given an eye point,
718         // a point to lookat, and a direction for which way is up. Here, we set the
719         // eye five units back along the z-axis and up three units, look at the
720         // origin, and define "up" to be in the y-direction.
721 
722         THR(m_pd3dDevice->SetTransform(D3DTS_VIEW,
723             rt::anon(rt::mat_look_at(D3DXVECTOR3(0, 0, -5)))));
724 
```

```
725     // Set the projection matrix
726     D3DXMATRIX matProj;
727     float aspect = ((float) m_d3dsdBackBuffer.Width)/m_d3dsdBackBuffer.Height;
728     ::D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI/4, aspect, 1.0f, 7.0f);
729     THR(m_pd3dDevice->SetTransform(D3DTS_PROJECTION, &matProj));
730
731     // Set up lighting states
732     D3DLIGHT9 light;
733     ::D3DUtil_InitLight(light, D3DLIGHT_DIRECTIONAL, -1.0f, -1.0f, 2.0f);
734     m_pd3dDevice->SetLight(0, &light);
735     m_pd3dDevice->LightEnable(0, TRUE);
736     m_pd3dDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
737
738     // create a tile in the same format as the back buffer and load it
739     THR(m_pd3dDevice->CreateOffscreenPlainSurface(TILE_SIZE, TILE_SIZE,
740         m_d3dsdBackBuffer.Format, D3DPOOL_DEFAULT, &m_device_tile, NULL));
741     THR(::D3DXLoadSurfaceFromSurface(m_device_tile, NULL, NULL,
742         m_tile, NULL, NULL, D3DX_FILTER_NONE, 0));
743
744     // rebuild stencil planes if needed
745     m_build_stencil = m_stencil_mask || m_stencil_stippling;
746
747     // build RECTs and POINTs for tiling the background
748     m_tile_rects.clear();
749     m_tile_offsets.clear();
750     const UINT width = m_d3dsdBackBuffer.Width;
751     const UINT height = m_d3dsdBackBuffer.Height;
752     UINT y;
753     for (y = 0; y < height; y += TILE_SIZE)
754     {
755         for (UINT x = 0; x < width; x += TILE_SIZE)
756         {
757             const bool y_even = ((y/TILE_SIZE) & 1) == 0;
758             const bool x_even = ((x/TILE_SIZE) & 1) == 0;
759
760             if ((x_even && y_even) || (!x_even && !y_even))
761             {
762                 const POINT offset = { x, y };
763                 const RECT rect =
764                 {
765                     0, 0,
766                     (x + TILE_SIZE) <= width ? TILE_SIZE : width-x,
767                     (y + TILE_SIZE) <= height ? TILE_SIZE : height-y
768                 };
769                 m_tile_rects.push_back(rect);
770                 m_tile_offsets.push_back(offset);
```

```
771             }
772         }
773     }
774
775     // build s_screen_vertex structures for stippling the window
776     m_stipple_verts = 0;
777     m_num_stipple_quads = (width + STIPPLE_SIZE - 1)/STIPPLE_SIZE*
778         (height + STIPPLE_SIZE - 1)/STIPPLE_SIZE;
779     THR(m_pd3dDevice->CreateVertexBuffer(m_num_stipple_quads*6*sizeof(s_screen_vertex),
780         D3DUSAGE_WRITEONLY, s_screen_vertex::FVF, D3DPOOL_MANAGED,
781         &m_stipple_verts, NULL));
782     rt::vertex_lock<s_screen_vertex> verts(m_stipple_verts);
783     s_screen_vertex *quad = verts.data();
784     for (y = 0; y < height; y += STIPPLE_SIZE)
785     {
786         for (UINT x = 0; x < width; x += STIPPLE_SIZE)
787         {
788             const float right = float(std::min(x + STIPPLE_SIZE, width));
789             const float bottom = float(std::min(y + STIPPLE_SIZE, height));
790             quad[0].x = x + 0.5f;
791             quad[0].y = y + 0.5f;
792             quad[0].z = 0.99f;
793             quad[0].rhw = 0.99f;
794             quad[0].u = 0;
795             quad[0].v = 1;
796
797             quad[1].x = right + 0.5f;
798             quad[1].y = y + 0.5f;
799             quad[1].z = 0.99f;
800             quad[1].rhw = 0.99f;
801             quad[1].u = 1;
802             quad[1].v = 1;
803
804             quad[2].x = x + 0.5f;
805             quad[2].y = bottom + 0.5f;
806             quad[2].z = 0.99f;
807             quad[2].rhw = 0.99f;
808             quad[2].u = 0;
809             quad[2].v = 0;
810
811             quad[3].x = x + 0.5f;
812             quad[3].y = bottom + 0.5f;
813             quad[3].z = 0.99f;
814             quad[3].rhw = 0.99f;
815             quad[3].u = 0;
816             quad[3].v = 0;
```

```
817
818                 quad[4].x = right + 0.5f;
819                 quad[4].y = y + 0.5f;
820                 quad[4].z = 0.99f;
821                 quad[4].rhw = 0.99f;
822                 quad[4].u = 1;
823                 quad[4].v = 1;
824
825                 quad[5].x = right + 0.5f;
826                 quad[5].y = bottom + 0.5f;
827                 quad[5].z = 0.99f;
828                 quad[5].rhw = 0.99f;
829                 quad[5].u = 1;
830                 quad[5].v = 0;
831
832                 quad += 6;
833             }
834         }
835
836         // Restore the font
837         THR(m_font.RestoreDeviceObjects());
838
839         return S_OK;
840     }
841
842
843
844
845     //-----------------------------------------------------------------------------
846     // Name: FrameMove()
847     // Desc: Called once per frame, the call is the entry point for animating
848     //       the scene.
849     //-----------------------------------------------------------------------------
850     HRESULT CMyD3DApplication::FrameMove()
851     {
852         // Update user input state
853         UpdateInput();
854
855         m_last_rot_x = m_rot_x;
856         m_last_rot_y = m_rot_y;
857
858         // Update the world state according to user input
859         if (m_animate_view)
860         {
861             m_rot_y += m_fElapsedTime;
862             if (m_rot_y > 2*D3DX_PI)
```

```
863           {
864               m_rot_y = std::fmodf(m_rot_y, 2*D3DX_PI);
865           }
866       }
867       else
868       {
869           if (m_input.m_left && !m_input.m_right)
870           {
871               m_rot_y += m_fElapsedTime;
872           }
873           else if (m_input.m_right && !m_input.m_left)
874           {
875               m_rot_y -= m_fElapsedTime;
876           }
877       }
878       if (m_input.m_up && !m_input.m_down)
879       {
880           m_rot_x += m_fElapsedTime;
881       }
882       else if (m_input.m_down && !m_input.m_up)
883       {
884           m_rot_x -= m_fElapsedTime;
885       }
886
887       return S_OK;
888   }
889
890
891
892
893   //-----------------------------------------------------------------------------
894   // Name: UpdateInput()
895   // Desc: Update the user input.  Called once per frame
896   //-----------------------------------------------------------------------------
897   void CMyD3DApplication::UpdateInput()
898   {
899       m_input.m_up    = (m_bActive && (GetAsyncKeyState(VK_UP)    & 0x8000) == 0x8000);
900       m_input.m_down  = (m_bActive && (GetAsyncKeyState(VK_DOWN)  & 0x8000) == 0x8000);
901       m_input.m_left  = (m_bActive && (GetAsyncKeyState(VK_LEFT)  & 0x8000) == 0x8000);
902       m_input.m_right = (m_bActive && (GetAsyncKeyState(VK_RIGHT) & 0x8000) == 0x8000);
903   }
904
905
906
907
908   //-----------------------------------------------------------------------------
```

```
909  // Name: Render()
910  // Desc: Called once per frame, the call is the entry point for 3d
911  //       rendering. This function sets up render states, clears the
912  //       viewport, and renders the scene.
913  //-----------------------------------------------------------------------------
914  HRESULT CMyD3DApplication::Render()
915  {
916      // Clear the viewport
917      THR(m_pd3dDevice->SetRenderState(D3DRS_SCISSORTESTENABLE, FALSE));
918      THR(m_pd3dDevice->Clear(0L, NULL, D3DCLEAR_TARGET |
919          D3DCLEAR_ZBUFFER, m_bg, 1.0f, 0L));
920
921      if (m_tile_background)
922      {
923          tile_background();
924      }
925
926      // enable or disable full-scene antialiasing
927      THR(m_pd3dDevice->SetRenderState(
928          D3DRS_MULTISAMPLEANTIALIAS,
929          m_multisample_antialias));
930
931      THR(m_pd3dDevice->BeginScene());
932
933      // set up state for rendering: materials & lighting
934      D3DMATERIAL9 mtrl;
935      ::D3DUtil_InitMaterial(mtrl, 1.0f, 0.0f, 0.0f);
936      mtrl.Diffuse = D3DXCOLOR(m_fg);
937      mtrl.Specular = D3DXCOLOR(m_specular);
938      THR(m_pd3dDevice->SetMaterial(&mtrl));
939
940      // build stencil planes, if necessary
941      if (m_build_stencil)
942      {
943          render_stencil();
944          m_build_stencil = false;
945      }
946
947      // texturing states
948      if (m_can_texture && m_textured)
949      {
950          const rt::s_tss ts_states[] =
951          {
952              D3DTSS_TEXCOORDINDEX,
953              D3DTSS_TCI_CAMERASPACEPOSITION,
954              D3DTSS_COLORARG1, D3DTA_TEXTURE,
```

```
955             D3DTSS_COLOROP, D3DTOP_MODULATE,
956             D3DTSS_COLORARG2, D3DTA_DIFFUSE,
957             D3DTSS_ALPHAARG1, D3DTA_TEXTURE,
958             D3DTSS_ALPHAOP, D3DTOP_SELECTARG2,
959             D3DTSS_ALPHAARG2, D3DTA_DIFFUSE,
960             D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2
961         };
962         rt::set_states(m_pd3dDevice, 0, ts_states,
963             NUM_OF(ts_states));
964         const rt::s_ss samp_states[] =
965         {
966             D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP,
967             D3DSAMP_ADDRESSV, D3DTADDRESS_WRAP,
968             D3DSAMP_MINFILTER, D3DTEXF_LINEAR,
969             D3DSAMP_MAGFILTER, D3DTEXF_LINEAR,
970             D3DSAMP_MIPFILTER, D3DTEXF_LINEAR
971         };
972         rt::set_states(m_pd3dDevice, 0, samp_states,
973             NUM_OF(samp_states));
974         ATLASSERT(m_texture);
975         THR(m_pd3dDevice->SetTexture(0, m_texture));
976         // set a (u,v) transform that makes the flag
977         // rectangular and positioned nicely on the
978         // teapot.  Invert the y-axis to get it in
979         // the proper vertical orientation.  Rotate
980         // it around the Z axis to make it a little
981         // more interesting.
982         D3DXMATRIX texform =
983             rt::mat_rot_z(D3DX_PI/4.0f)*
984             rt::mat_scale(0.5f, -1, 1)*
985             rt::mat_trans(-0.6f, -0.5f, 0);
986         THR(m_pd3dDevice->SetTransform(D3DTS_TEXTURE0,
987             &texform));
988     }
989     else
990     {
991         // disable texturing
992         const rt::s_tss ts_states[] =
993         {
994             D3DTSS_TEXCOORDINDEX, 0,
995             D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_DISABLE,
996             D3DTSS_COLOROP, D3DTOP_DISABLE,
997             D3DTSS_ALPHAOP, D3DTOP_DISABLE
998         };
999         rt::set_states(m_pd3dDevice, 0, ts_states,
1000            NUM_OF(ts_states));
```

```
1001            THR(m_pd3dDevice->SetTexture(0, NULL));
1002        }
1003
1004        // stencil test state
1005        if (m_stencil_mask || m_stencil_stippling)
1006        {
1007            // draw teapot only where no stencil bits set,
1008            // and don't modify the stencil bits
1009            rt::s_rs states[] =
1010            {
1011                D3DRS_STENCILENABLE, true,
1012                D3DRS_STENCILFUNC, D3DCMP_EQUAL,
1013                D3DRS_STENCILREF, 0,
1014                D3DRS_STENCILMASK, ~0UL,
1015                D3DRS_STENCILWRITEMASK, 0,
1016                D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP,
1017                D3DRS_STENCILPASS, D3DSTENCILOP_KEEP,
1018                D3DRS_STENCILZFAIL, D3DSTENCILOP_KEEP
1019            };
1020            rt::set_states(m_pd3dDevice, states,
1021                NUM_OF(states));
1022        }
1023        else
1024        {
1025            THR(m_pd3dDevice->SetRenderState(
1026                D3DRS_STENCILENABLE, false));
1027        }
1028
1029        // setup blending state
1030        {
1031            rt::s_rs states[] =
1032            {
1033                D3DRS_ALPHABLENDENABLE, m_color_blend_enable,
1034                D3DRS_SRCBLEND, m_color_src_blend,
1035                D3DRS_DESTBLEND, m_color_dest_blend,
1036                D3DRS_BLENDOP, m_color_blend_op,
1037                D3DRS_BLENDFACTOR, m_blend_factor,
1038                D3DRS_SEPARATEALPHABLENDENABLE, m_alpha_blend_enable,
1039                D3DRS_SRCBLENDALPHA, m_alpha_src_blend,
1040                D3DRS_DESTBLENDALPHA, m_alpha_dest_blend,
1041                D3DRS_BLENDOPALPHA, m_alpha_blend_op
1042            };
1043            rt::set_states(m_pd3dDevice, states,
1044                NUM_OF(states));
1045        }
1046
```

```
1047        // frame buffer render states
1048        const rt::s_rs states[] =
1049        {
1050            D3DRS_SCISSORTESTENABLE, m_scissor_enable,
1051            D3DRS_NORMALIZENORMALS, true,
1052            D3DRS_LIGHTING, true,
1053            D3DRS_SPECULARENABLE, true,
1054            D3DRS_ZENABLE, true,
1055            D3DRS_ZFUNC, D3DCMP_LESS,
1056            D3DRS_DITHERENABLE, m_dithered,
1057            D3DRS_ZWRITEENABLE, m_z_write_enable
1058        };
1059        rt::set_states(m_pd3dDevice, states, NUM_OF(states));
1060
1061        // Render the teapot mesh
1062        if (m_multisample_depth_of_field)
1063        {
1064            render_depth_of_field();
1065        }
1066        else if (m_multisample_motion_blur)
1067        {
1068            render_motion_blur();
1069        }
1070        else
1071        {
1072            render_teapot(rt::mat_rot_x(m_rot_x)*
1073                          rt::mat_rot_y(m_rot_y));
1074        }
1075
1076        // Render stats and help text
1077        if (m_show_stats)
1078        {
1079            RenderText();
1080        }
1081
1082        THR(m_pd3dDevice->EndScene());
1083
1084        return S_OK;
1085    }
1086
1087
1088    //////////////////////////////////////////////////////////////////////
1089    // operator<<(e_composite_operator)
1090    //
1091    // Formats an enumerant of e_composite_operator as a string.
1092    //
```

```
1093  rt::tostream &
1094  operator<<(rt::tostream &stream, e_composite_operator blending)
1095  {
1096      switch (blending)
1097      {
1098  #define BLEND_OUT(enumerant_, str_) \
1099      case enumerant_: stream << _T(str_); break
1100          BLEND_OUT(E_COMPOSITE_CLEAR, "clear");
1101          BLEND_OUT(E_COMPOSITE_SRC, "src");
1102          BLEND_OUT(E_COMPOSITE_DEST, "dest");
1103          BLEND_OUT(E_COMPOSITE_SRC_OVER_DEST, "src over dest");
1104          BLEND_OUT(E_COMPOSITE_DEST_OVER_SRC, "dest over src");
1105          BLEND_OUT(E_COMPOSITE_SRC_IN_DEST, "src in dest");
1106          BLEND_OUT(E_COMPOSITE_DEST_IN_SRC, "dest in src");
1107          BLEND_OUT(E_COMPOSITE_SRC_ATOP_DEST, "src atop dest");
1108          BLEND_OUT(E_COMPOSITE_DEST_ATOP_SRC, "dest atop src");
1109          BLEND_OUT(E_COMPOSITE_SRC_OUT_DEST, "src out dest");
1110          BLEND_OUT(E_COMPOSITE_DEST_OUT_SRC, "dest out src");
1111          BLEND_OUT(E_COMPOSITE_SRC_XOR_DEST, "src xor dest");
1112          BLEND_OUT(E_COMPOSITE_NONE, "none");
1113  #undef BLEND_OUT
1114      default:
1115          ATLASSERT(false);
1116      }
1117      return stream;
1118  }
1119
1120  //-----------------------------------------------------------------------------
1121  // Name: RenderText()
1122  // Desc: Renders stats and help text to the scene.
1123  //-----------------------------------------------------------------------------
1124  HRESULT CMyD3DApplication::RenderText()
1125  {
1126      THR(m_pd3dDevice->SetRenderState(D3DRS_SCISSORTESTENABLE, false));
1127
1128      // render text with a drop shadow
1129      render_text_aux(1, 1, D3DCOLOR_XRGB(0, 0, 0));
1130      render_text_aux(0, 0, m_text_fg);
1131
1132      return S_OK;
1133  }
1134
1135
1136  //////////////////////////////////////////////////////////////////////
1137  // CMyD3DApplication::render_text_aux
1138  //
```

```
1139  // Render lines of text at a specific offset and in a specific color.
1140  // Some program options are displayed in addition to the standard
1141  // AppWizard sample text.
1142  //
1143  void
1144  CMyD3DApplication::render_text_aux(float offset_x, float offset_y, D3DCOLOR fg)
1145  {
1146      // Output display stats
1147      float y = 2.0f + offset_y;
1148      float x = 2.0f + offset_x;
1149      rt::tostringstream msg;
1150
1151      text(x, y, fg, m_strFrameStats);
1152      y += 20.0f; text(x, y, fg, m_strDeviceStats);
1153
1154      msg << (m_destination_alpha ?
1155          _T("Destination alpha") : _T("No destination alpha"));
1156      if (m_dithered)
1157      {
1158          msg << _T(", dither");
1159      }
1160      if (E_COMPOSITE_NONE != m_color_composite)
1161      {
1162          msg << _T(", ") << m_color_composite;
1163      }
1164      if (m_multisample_antialias)
1165      {
1166          msg << _T(", multisample antialias");
1167      }
1168      else if (m_multisample_depth_of_field)
1169      {
1170          msg << _T(", multisample depth of field");
1171      }
1172      else if (m_multisample_motion_blur)
1173      {
1174          msg << _T(", multisample motion blur");
1175      }
1176      if (m_stencil_mask)
1177      {
1178          msg << _T(", stencil mask");
1179      }
1180      if (m_stencil_stippling)
1181      {
1182          msg << _T(", stencil stipple");
1183      }
1184      if (m_scissor_enable)
```

```
1185        {
1186            msg << _T(", scissor");
1187        }
1188        msg << std::ends;
1189        y += 20.0f; text(x, y, fg, msg);
1190
1191        if (m_color_blend_enable &&
1192            ((D3DBLEND_BLENDFACTOR == m_color_src_blend) ||
1193             (D3DBLEND_BLENDFACTOR == m_color_dest_blend) ||
1194             (D3DBLEND_INVBLENDFACTOR == m_color_src_blend) ||
1195             (D3DBLEND_INVBLENDFACTOR == m_color_dest_blend) ||
1196             (m_alpha_blend_enable &&
1197              (D3DBLEND_BLENDFACTOR == m_alpha_src_blend) ||
1198              (D3DBLEND_BLENDFACTOR == m_alpha_dest_blend) ||
1199              (D3DBLEND_INVBLENDFACTOR == m_alpha_src_blend) ||
1200              (D3DBLEND_INVBLENDFACTOR == m_alpha_dest_blend))))
1201        {
1202            msg.seekp(0);
1203            msg << _T("Blend factor: RGBA(") << int(rt::D3DColor_Red(m_blend_factor)
1204                << _T(", ") << int(rt::D3DColor_Green(m_blend_factor))
1205                << _T(", ") << int(rt::D3DColor_Blue(m_blend_factor))
1206                << _T(", ") << int(rt::D3DColor_Alpha(m_blend_factor))
1207                << _T(")") << std::ends;
1208            y += 20.0f; text(x, y, fg, msg);
1209        }
1210
1211
1212        // Output statistics & help
1213        y = float(m_d3dsdBackBuffer.Height + offset_y + 2);
1214        msg.seekp(0);
1215        msg << _T("Arrow keys: Up=") << int(m_input.m_up) << _T(" Down=") <<
1216            int(m_input.m_down) << _T(" Left=") << int(m_input.m_left) <<
1217            _T(" Right=") << int(m_input.m_right) << std::ends;
1218        y -= 20.0f; text(x, y, fg, msg);
1219
1220        y -= 20.0f; text(x, y, fg, _T("Use arrow keys to rotate object"));
1221        y -= 20.0f; text(x, y, fg, _T("Press 'F2' to configure display"));
1222    }
1223
1224
1225    //-----------------------------------------------------------------------------
1226    // Name: MsgProc()
1227    // Desc: Overrrides the main WndProc, so the sample can do custom message
1228    //       handling (e.g. processing mouse, keyboard, or menu commands).
1229    //-----------------------------------------------------------------------------
1230    LRESULT CMyD3DApplication::MsgProc(HWND hWnd, UINT msg, WPARAM wParam,
```

```
1231                                        LPARAM lParam)
1232  {
1233      LRESULT result = 0;
1234      bool handled = false;
1235
1236      switch (msg)
1237      {
1238      case WM_LBUTTONDOWN:
1239          result = on_left_button_down(hWnd, wParam, lParam, handled);
1240          break;
1241
1242      case WM_MOUSEMOVE:
1243          result = on_mouse_move(hWnd, wParam, lParam, handled);
1244          break;
1245
1246      case WM_LBUTTONUP:
1247          result = on_left_button_up(hWnd, wParam, lParam, handled);
1248          break;
1249
1250      case WM_PAINT:
1251          if (m_bLoadingApp)
1252          {
1253              // Draw on the window tell the user that the app is loading
1254              HDC hDC = TWS(::GetDC(hWnd));
1255              RECT rct;
1256              TWS(::GetClientRect(hWnd, &rct));
1257              ::DrawText(hDC, _T("Loading... Please wait"), -1, &rct,
1258                  DT_CENTER | DT_VCENTER | DT_SINGLELINE);
1259              TWS(::ReleaseDC(hWnd, hDC));
1260          }
1261          break;
1262
1263      case WM_COMMAND:
1264          result = on_command(hWnd, wParam, lParam, handled);
1265          break;
1266      }
1267
1268      return handled ? result :
1269          CD3DApplication::MsgProc(hWnd, msg, wParam, lParam);
1270  }
1271
1272
1273
1274
1275  //-----------------------------------------------------------------------------
1276  // Name: InvalidateDeviceObjects()
```

```
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
1285      return S_OK;
1286  }
1287
1288
1289
1290
1291  //-----------------------------------------------------------------------------
1292  // Name: DeleteDeviceObjects()
1293  // Desc: Paired with InitDeviceObjects()
1294  //       Called when the app is exiting, or the device is being changed,
1295  //       this function deletes any device dependent objects.
1296  //-----------------------------------------------------------------------------
1297  HRESULT CMyD3DApplication::DeleteDeviceObjects()
1298  {
1299      m_teapot = 0;
1300      m_torus = 0;
1301      m_texture = 0;
1302      m_tile = 0;
1303      m_device_tile = 0;
1304      m_stipple = 0;
1305      THR(m_font.DeleteDeviceObjects());
1306
1307      return S_OK;
1308  }
1309
1310
1311
1312
1313  //-----------------------------------------------------------------------------
1314  // Name: FinalCleanup()
1315  // Desc: Paired with OneTimeSceneInit()
1316  //       Called before the app exits, this function gives the app the chance
1317  //       to cleanup after itself.
1318  //-----------------------------------------------------------------------------
1319  HRESULT CMyD3DApplication::FinalCleanup()
1320  {
1321      // TODO: Perform any final cleanup needed
1322
```

```
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        if (SUCCEEDED(hr))
1347        {
1348            m_texture = new_texture;
1349            if (file)
1350            {
1351                m_texture_file = file;
1352            }
1353        }
1354        else
1355        {
1356            THR(hr);
1357        }
1358    }
1359
1360    template <typename T>
1361    DWORD
1362    find_id(const rt::s_enum_value<T> *items, UINT num_items, T value)
1363    {
1364        for (UINT i = 0; i < num_items; i++)
1365        {
1366            if (value == items[i].m_state)
1367            {
1368                return items[i].m_value;
```

```
1369            }
1370        }
1371        ATLASSERT(false);
1372        return 0;
1373    }
1374
1375    DWORD
1376    color_composite_id(e_composite_operator value)
1377    {
1378        const rt::s_enum_value<e_composite_operator> items[] =
1379        {
1380            E_COMPOSITE_CLEAR,          IDM_COLOR_COMPOSITE_CLEAR,
1381            E_COMPOSITE_SRC,            IDM_COLOR_COMPOSITE_SRC,
1382            E_COMPOSITE_DEST,           IDM_COLOR_COMPOSITE_DEST,
1383            E_COMPOSITE_SRC_OVER_DEST,  IDM_COLOR_COMPOSITE_SRC_OVER_DEST,
1384            E_COMPOSITE_DEST_OVER_SRC,  IDM_COLOR_COMPOSITE_DEST_OVER_SRC,
1385            E_COMPOSITE_SRC_IN_DEST,    IDM_COLOR_COMPOSITE_SRC_IN_DEST,
1386            E_COMPOSITE_DEST_IN_SRC,    IDM_COLOR_COMPOSITE_DEST_IN_SRC,
1387            E_COMPOSITE_SRC_ATOP_DEST,  IDM_COLOR_COMPOSITE_SRC_ATOP_DEST,
1388            E_COMPOSITE_DEST_ATOP_SRC,  IDM_COLOR_COMPOSITE_DEST_ATOP_SRC,
1389            E_COMPOSITE_SRC_OUT_DEST,   IDM_COLOR_COMPOSITE_SRC_OUT_DEST,
1390            E_COMPOSITE_DEST_OUT_SRC,   IDM_COLOR_COMPOSITE_DEST_OUT_SRC,
1391            E_COMPOSITE_SRC_XOR_DEST,   IDM_COLOR_COMPOSITE_SRC_XOR_DEST
1392        };
1393        return find_id(items, NUM_OF(items), value);
1394    }
1395
1396    DWORD
1397    color_blend_op_id(D3DBLENDOP value)
1398    {
1399        const rt::s_enum_value<D3DBLENDOP> items[] =
1400        {
1401            D3DBLENDOP_ADD,             IDM_COLOR_BLEND_OP_ADD,
1402            D3DBLENDOP_SUBTRACT,        IDM_COLOR_BLEND_OP_SUBTRACT,
1403            D3DBLENDOP_REVSUBTRACT,     IDM_COLOR_BLEND_OP_REVERSE_SUBTRACT,
1404            D3DBLENDOP_MIN,             IDM_COLOR_BLEND_OP_MINIMUM,
1405            D3DBLENDOP_MAX,             IDM_COLOR_BLEND_OP_MAXIMUM
1406        };
1407        return find_id(items, NUM_OF(items), value);
1408    }
1409
1410    DWORD
1411    color_src_blend_id(D3DBLEND value)
1412    {
1413        const rt::s_enum_value<D3DBLEND> items[] =
1414        {
```

```
1415            D3DBLEND_ZERO,              IDM_COLOR_SRC_BLEND_ZERO,
1416            D3DBLEND_ONE,               IDM_COLOR_SRC_BLEND_ONE,
1417            D3DBLEND_SRCCOLOR,          IDM_COLOR_SRC_BLEND_SRC_COLOR,
1418            D3DBLEND_INVSRCCOLOR,       IDM_COLOR_SRC_BLEND_INV_SRC_COLOR,
1419            D3DBLEND_SRCALPHA,          IDM_COLOR_SRC_BLEND_SRC_ALPHA,
1420            D3DBLEND_INVSRCALPHA,       IDM_COLOR_SRC_BLEND_INV_SRC_ALPHA,
1421            D3DBLEND_DESTALPHA,         IDM_COLOR_SRC_BLEND_DEST_ALPHA,
1422            D3DBLEND_INVDESTALPHA,      IDM_COLOR_SRC_BLEND_INV_DEST_ALPHA,
1423            D3DBLEND_DESTCOLOR,         IDM_COLOR_SRC_BLEND_DEST_COLOR,
1424            D3DBLEND_INVDESTCOLOR,      IDM_COLOR_SRC_BLEND_INV_DEST_COLOR,
1425            D3DBLEND_SRCALPHASAT,       IDM_COLOR_SRC_BLEND_SRC_ALPHA_SAT,
1426            D3DBLEND_BOTHINVSRCALPHA,   IDM_COLOR_SRC_BLEND_BOTH_INV_SRC_ALPHA,
1427            D3DBLEND_BLENDFACTOR,       IDM_COLOR_SRC_BLEND_FACTOR,
1428            D3DBLEND_INVBLENDFACTOR,    IDM_COLOR_SRC_BLEND_INV_FACTOR
1429        };
1430        return find_id(items, NUM_OF(items), value);
1431    }
1432
1433    DWORD
1434    color_dest_blend_id(D3DBLEND value)
1435    {
1436        const rt::s_enum_value<D3DBLEND> items[] =
1437        {
1438            D3DBLEND_ZERO,              IDM_COLOR_DEST_BLEND_ZERO,
1439            D3DBLEND_ONE,               IDM_COLOR_DEST_BLEND_ONE,
1440            D3DBLEND_SRCCOLOR,          IDM_COLOR_DEST_BLEND_SRC_COLOR,
1441            D3DBLEND_INVSRCCOLOR,       IDM_COLOR_DEST_BLEND_INV_SRC_COLOR,
1442            D3DBLEND_SRCALPHA,          IDM_COLOR_DEST_BLEND_SRC_ALPHA,
1443            D3DBLEND_INVSRCALPHA,       IDM_COLOR_DEST_BLEND_INV_SRC_ALPHA,
1444            D3DBLEND_DESTALPHA,         IDM_COLOR_DEST_BLEND_DEST_ALPHA,
1445            D3DBLEND_INVDESTALPHA,      IDM_COLOR_DEST_BLEND_INV_DEST_ALPHA,
1446            D3DBLEND_DESTCOLOR,         IDM_COLOR_DEST_BLEND_DEST_COLOR,
1447            D3DBLEND_INVDESTCOLOR,      IDM_COLOR_DEST_BLEND_INV_DEST_COLOR,
1448            D3DBLEND_SRCALPHASAT,       IDM_COLOR_DEST_BLEND_SRC_ALPHA_SAT,
1449            D3DBLEND_BLENDFACTOR,       IDM_COLOR_DEST_BLEND_FACTOR,
1450            D3DBLEND_INVBLENDFACTOR,    IDM_COLOR_DEST_BLEND_INV_FACTOR
1451        };
1452        return find_id(items, NUM_OF(items), value);
1453    }
1454
1455    DWORD
1456    alpha_composite_id(e_composite_operator value)
1457    {
1458        const rt::s_enum_value<e_composite_operator> items[] =
1459        {
1460            E_COMPOSITE_CLEAR,          IDM_ALPHA_COMPOSITE_CLEAR,
```

```
1461            E_COMPOSITE_SRC,            IDM_ALPHA_COMPOSITE_SRC,
1462            E_COMPOSITE_DEST,           IDM_ALPHA_COMPOSITE_DEST,
1463            E_COMPOSITE_SRC_OVER_DEST,  IDM_ALPHA_COMPOSITE_SRC_OVER_DEST,
1464            E_COMPOSITE_DEST_OVER_SRC,  IDM_ALPHA_COMPOSITE_DEST_OVER_SRC,
1465            E_COMPOSITE_SRC_IN_DEST,    IDM_ALPHA_COMPOSITE_SRC_IN_DEST,
1466            E_COMPOSITE_DEST_IN_SRC,    IDM_ALPHA_COMPOSITE_DEST_IN_SRC,
1467            E_COMPOSITE_SRC_ATOP_DEST,  IDM_ALPHA_COMPOSITE_SRC_ATOP_DEST,
1468            E_COMPOSITE_DEST_ATOP_SRC,  IDM_ALPHA_COMPOSITE_DEST_ATOP_SRC,
1469            E_COMPOSITE_SRC_OUT_DEST,   IDM_ALPHA_COMPOSITE_SRC_OUT_DEST,
1470            E_COMPOSITE_DEST_OUT_SRC,   IDM_ALPHA_COMPOSITE_DEST_OUT_SRC,
1471            E_COMPOSITE_SRC_XOR_DEST,   IDM_ALPHA_COMPOSITE_SRC_XOR_DEST
1472        };
1473        return find_id(items, NUM_OF(items), value);
1474    }
1475
1476    DWORD
1477    alpha_blend_op_id(D3DBLENDOP value)
1478    {
1479        const rt::s_enum_value<D3DBLENDOP> items[] =
1480        {
1481            D3DBLENDOP_ADD,             IDM_ALPHA_BLEND_OP_ADD,
1482            D3DBLENDOP_SUBTRACT,        IDM_ALPHA_BLEND_OP_SUBTRACT,
1483            D3DBLENDOP_REVSUBTRACT,     IDM_ALPHA_BLEND_OP_REVERSE_SUBTRACT,
1484            D3DBLENDOP_MIN,             IDM_ALPHA_BLEND_OP_MINIMUM,
1485            D3DBLENDOP_MAX,             IDM_ALPHA_BLEND_OP_MAXIMUM
1486        };
1487        return find_id(items, NUM_OF(items), value);
1488    }
1489
1490    DWORD
1491    alpha_src_blend_id(D3DBLEND value)
1492    {
1493        const rt::s_enum_value<D3DBLEND> items[] =
1494        {
1495            D3DBLEND_ZERO,              IDM_ALPHA_SRC_BLEND_ZERO,
1496            D3DBLEND_ONE,               IDM_ALPHA_SRC_BLEND_ONE,
1497            D3DBLEND_SRCCOLOR,          IDM_ALPHA_SRC_BLEND_SRC_COLOR,
1498            D3DBLEND_INVSRCCOLOR,       IDM_ALPHA_SRC_BLEND_INV_SRC_COLOR,
1499            D3DBLEND_SRCALPHA,          IDM_ALPHA_SRC_BLEND_SRC_ALPHA,
1500            D3DBLEND_INVSRCALPHA,       IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA,
1501            D3DBLEND_DESTALPHA,         IDM_ALPHA_SRC_BLEND_DEST_ALPHA,
1502            D3DBLEND_INVDESTALPHA,      IDM_ALPHA_SRC_BLEND_INV_DEST_ALPHA,
1503            D3DBLEND_DESTCOLOR,         IDM_ALPHA_SRC_BLEND_DEST_COLOR,
1504            D3DBLEND_INVDESTCOLOR,      IDM_ALPHA_SRC_BLEND_INV_DEST_COLOR,
1505            D3DBLEND_SRCALPHASAT,       IDM_ALPHA_SRC_BLEND_SRC_ALPHA_SAT,
1506            D3DBLEND_BOTHINVSRCALPHA,   IDM_ALPHA_SRC_BLEND_BOTH_INV_SRC_ALPHA,
```

```
1507            D3DBLEND_BLENDFACTOR,       IDM_ALPHA_SRC_BLEND_FACTOR,
1508            D3DBLEND_INVBLENDFACTOR,    IDM_ALPHA_SRC_BLEND_INV_FACTOR
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_BLEND_ZERO,
1519            D3DBLEND_ONE,               IDM_ALPHA_DEST_BLEND_ONE,
1520            D3DBLEND_SRCCOLOR,          IDM_ALPHA_DEST_BLEND_SRC_COLOR,
1521            D3DBLEND_INVSRCCOLOR,       IDM_ALPHA_DEST_BLEND_INV_SRC_COLOR,
1522            D3DBLEND_SRCALPHA,          IDM_ALPHA_DEST_BLEND_SRC_ALPHA,
1523            D3DBLEND_INVSRCALPHA,       IDM_ALPHA_DEST_BLEND_INV_SRC_ALPHA,
1524            D3DBLEND_DESTALPHA,         IDM_ALPHA_DEST_BLEND_DEST_ALPHA,
1525            D3DBLEND_INVDESTALPHA,      IDM_ALPHA_DEST_BLEND_INV_DEST_ALPHA,
1526            D3DBLEND_DESTCOLOR,         IDM_ALPHA_DEST_BLEND_DEST_COLOR,
1527            D3DBLEND_INVDESTCOLOR,      IDM_ALPHA_DEST_BLEND_INV_DEST_COLOR,
1528            D3DBLEND_SRCALPHASAT,       IDM_ALPHA_DEST_BLEND_SRC_ALPHA_SAT,
1529            D3DBLEND_BLENDFACTOR,       IDM_ALPHA_DEST_BLEND_FACTOR,
1530            D3DBLEND_INVBLENDFACTOR,    IDM_ALPHA_DEST_BLEND_INV_FACTOR
1531        };
1532        return find_id(items, NUM_OF(items), value);
1533    }
1534
1535    void
1536    CMyD3DApplication::update_blending(HMENU menu, bool checked)
1537    {
1538        if (! checked)
1539        {
1540            // enable all the menu items first, so that when we're called
1541            // again with checked==false, we can disable only those that
1542            // need to be disabled.
1543            const DWORD ids[] =
1544            {
1545                IDM_COLOR_COMPOSITE_CLEAR,
1546                IDM_COLOR_COMPOSITE_SRC,
1547                IDM_COLOR_COMPOSITE_DEST,
1548                IDM_COLOR_COMPOSITE_SRC_OVER_DEST,
1549                IDM_COLOR_COMPOSITE_DEST_OVER_SRC,
1550                IDM_COLOR_COMPOSITE_SRC_IN_DEST,
1551                IDM_COLOR_COMPOSITE_DEST_IN_SRC,
1552                IDM_COLOR_COMPOSITE_SRC_ATOP_DEST,
```

```
1553            IDM_COLOR_COMPOSITE_DEST_ATOP_SRC,
1554            IDM_COLOR_COMPOSITE_SRC_OUT_DEST,
1555            IDM_COLOR_COMPOSITE_DEST_OUT_SRC,
1556            IDM_COLOR_COMPOSITE_SRC_XOR_DEST,
1557
1558            IDM_COLOR_BLEND_OP_ADD,
1559            IDM_COLOR_BLEND_OP_SUBTRACT,
1560            IDM_COLOR_BLEND_OP_REVERSE_SUBTRACT,
1561            IDM_COLOR_BLEND_OP_MINIMUM,
1562            IDM_COLOR_BLEND_OP_MAXIMUM,
1563
1564            IDM_COLOR_SRC_BLEND_ZERO,
1565            IDM_COLOR_SRC_BLEND_ONE,
1566            IDM_COLOR_SRC_BLEND_SRC_COLOR,
1567            IDM_COLOR_SRC_BLEND_INV_SRC_COLOR,
1568            IDM_COLOR_SRC_BLEND_SRC_ALPHA,
1569            IDM_COLOR_SRC_BLEND_INV_SRC_ALPHA,
1570            IDM_COLOR_SRC_BLEND_DEST_ALPHA,
1571            IDM_COLOR_SRC_BLEND_INV_DEST_ALPHA,
1572            IDM_COLOR_SRC_BLEND_DEST_COLOR,
1573            IDM_COLOR_SRC_BLEND_INV_DEST_COLOR,
1574            IDM_COLOR_SRC_BLEND_SRC_ALPHA_SAT,
1575            IDM_COLOR_SRC_BLEND_BOTH_INV_SRC_ALPHA,
1576            IDM_COLOR_SRC_BLEND_FACTOR,
1577            IDM_COLOR_SRC_BLEND_INV_FACTOR,
1578
1579            IDM_COLOR_DEST_BLEND_ZERO,
1580            IDM_COLOR_DEST_BLEND_ONE,
1581            IDM_COLOR_DEST_BLEND_SRC_COLOR,
1582            IDM_COLOR_DEST_BLEND_INV_SRC_COLOR,
1583            IDM_COLOR_DEST_BLEND_SRC_ALPHA,
1584            IDM_COLOR_DEST_BLEND_INV_SRC_ALPHA,
1585            IDM_COLOR_DEST_BLEND_DEST_ALPHA,
1586            IDM_COLOR_DEST_BLEND_INV_DEST_ALPHA,
1587            IDM_COLOR_DEST_BLEND_DEST_COLOR,
1588            IDM_COLOR_DEST_BLEND_INV_DEST_COLOR,
1589            IDM_COLOR_DEST_BLEND_SRC_ALPHA_SAT,
1590            IDM_COLOR_DEST_BLEND_FACTOR,
1591            IDM_COLOR_DEST_BLEND_INV_FACTOR,
1592
1593            IDM_ALPHA_COMPOSITE_CLEAR,
1594            IDM_ALPHA_COMPOSITE_SRC,
1595            IDM_ALPHA_COMPOSITE_DEST,
1596            IDM_ALPHA_COMPOSITE_SRC_OVER_DEST,
1597            IDM_ALPHA_COMPOSITE_DEST_OVER_SRC,
1598            IDM_ALPHA_COMPOSITE_SRC_IN_DEST,
```

```
1599                IDM_ALPHA_COMPOSITE_DEST_IN_SRC,
1600                IDM_ALPHA_COMPOSITE_SRC_ATOP_DEST,
1601                IDM_ALPHA_COMPOSITE_DEST_ATOP_SRC,
1602                IDM_ALPHA_COMPOSITE_SRC_OUT_DEST,
1603                IDM_ALPHA_COMPOSITE_DEST_OUT_SRC,
1604                IDM_ALPHA_COMPOSITE_SRC_XOR_DEST,
1605
1606                IDM_ALPHA_BLEND_OP_ADD,
1607                IDM_ALPHA_BLEND_OP_SUBTRACT,
1608                IDM_ALPHA_BLEND_OP_REVERSE_SUBTRACT,
1609                IDM_ALPHA_BLEND_OP_MINIMUM,
1610                IDM_ALPHA_BLEND_OP_MAXIMUM,
1611
1612                IDM_ALPHA_SRC_BLEND_ZERO,
1613                IDM_ALPHA_SRC_BLEND_ONE,
1614                IDM_ALPHA_SRC_BLEND_SRC_COLOR,
1615                IDM_ALPHA_SRC_BLEND_INV_SRC_COLOR,
1616                IDM_ALPHA_SRC_BLEND_SRC_ALPHA,
1617                IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA,
1618                IDM_ALPHA_SRC_BLEND_DEST_ALPHA,
1619                IDM_ALPHA_SRC_BLEND_INV_DEST_ALPHA,
1620                IDM_ALPHA_SRC_BLEND_DEST_COLOR,
1621                IDM_ALPHA_SRC_BLEND_INV_DEST_COLOR,
1622                IDM_ALPHA_SRC_BLEND_SRC_ALPHA_SAT,
1623                IDM_ALPHA_SRC_BLEND_BOTH_INV_SRC_ALPHA,
1624                IDM_ALPHA_SRC_BLEND_FACTOR,
1625                IDM_ALPHA_SRC_BLEND_INV_FACTOR,
1626
1627                IDM_ALPHA_DEST_BLEND_ZERO,
1628                IDM_ALPHA_DEST_BLEND_ONE,
1629                IDM_ALPHA_DEST_BLEND_SRC_COLOR,
1630                IDM_ALPHA_DEST_BLEND_INV_SRC_COLOR,
1631                IDM_ALPHA_DEST_BLEND_SRC_ALPHA,
1632                IDM_ALPHA_DEST_BLEND_INV_SRC_ALPHA,
1633                IDM_ALPHA_DEST_BLEND_DEST_ALPHA,
1634                IDM_ALPHA_DEST_BLEND_INV_DEST_ALPHA,
1635                IDM_ALPHA_DEST_BLEND_DEST_COLOR,
1636                IDM_ALPHA_DEST_BLEND_INV_DEST_COLOR,
1637                IDM_ALPHA_DEST_BLEND_SRC_ALPHA_SAT,
1638                IDM_ALPHA_DEST_BLEND_FACTOR,
1639                IDM_ALPHA_DEST_BLEND_INV_FACTOR
1640            };
1641            for (UINT i = 0; i < NUM_OF(ids); i++)
1642            {
1643                rt::enable_menu(menu, ids[i], true);
1644            }
```

```
1645        }
1646        else
1647        {
1648            // if we don't have destination alpha, we might have to change the
1649            // blending selection
1650            if (!m_destination_alpha)
1651            {
1652                if (!m_color_allow_dest_alpha)
1653                {
1654                    switch (m_color_composite)
1655                    {
1656                    case E_COMPOSITE_DEST:
1657                    case E_COMPOSITE_DEST_OVER_SRC:
1658                    case E_COMPOSITE_SRC_IN_DEST:
1659                    case E_COMPOSITE_SRC_ATOP_DEST:
1660                    case E_COMPOSITE_DEST_ATOP_SRC:
1661                    case E_COMPOSITE_SRC_OUT_DEST:
1662                    case E_COMPOSITE_SRC_XOR_DEST:
1663                        m_color_composite = E_COMPOSITE_NONE;
1664                        m_color_src_blend = D3DBLEND_ONE;
1665                        m_color_dest_blend = D3DBLEND_ZERO;
1666                        break;
1667                    }
1668                    switch (m_color_src_blend)
1669                    {
1670                    case D3DBLEND_DESTALPHA:
1671                    case D3DBLEND_INVDESTALPHA:
1672                    case D3DBLEND_DESTCOLOR:
1673                    case D3DBLEND_INVDESTCOLOR:
1674                        m_color_src_blend = D3DBLEND_ONE;
1675                        break;
1676                    }
1677                    switch (m_color_dest_blend)
1678                    {
1679                    case D3DBLEND_DESTALPHA:
1680                    case D3DBLEND_INVDESTALPHA:
1681                    case D3DBLEND_DESTCOLOR:
1682                    case D3DBLEND_INVDESTCOLOR:
1683                        m_color_dest_blend = D3DBLEND_ZERO;
1684                        break;
1685                    }
1686                }
1687                if (!m_alpha_allow_dest_alpha)
1688                {
1689                    switch (m_alpha_composite)
1690                    {
```

```
1691                    case E_COMPOSITE_DEST:
1692                    case E_COMPOSITE_DEST_OVER_SRC:
1693                    case E_COMPOSITE_SRC_IN_DEST:
1694                    case E_COMPOSITE_SRC_ATOP_DEST:
1695                    case E_COMPOSITE_DEST_ATOP_SRC:
1696                    case E_COMPOSITE_SRC_OUT_DEST:
1697                    case E_COMPOSITE_SRC_XOR_DEST:
1698                        m_alpha_composite = E_COMPOSITE_NONE;
1699                        m_alpha_src_blend = D3DBLEND_ONE;
1700                        m_alpha_dest_blend = D3DBLEND_ZERO;
1701                        break;
1702                    }
1703                    switch (m_alpha_src_blend)
1704                    {
1705                    case D3DBLEND_DESTALPHA:
1706                    case D3DBLEND_INVDESTALPHA:
1707                    case D3DBLEND_DESTCOLOR:
1708                    case D3DBLEND_INVDESTCOLOR:
1709                        m_alpha_src_blend = D3DBLEND_ONE;
1710                        break;
1711                    }
1712                    switch (m_alpha_dest_blend)
1713                    {
1714                    case D3DBLEND_DESTALPHA:
1715                    case D3DBLEND_INVDESTALPHA:
1716                    case D3DBLEND_DESTCOLOR:
1717                    case D3DBLEND_INVDESTCOLOR:
1718                        m_alpha_dest_blend = D3DBLEND_ZERO;
1719                        break;
1720                    }
1721                }
1722            }
1723
1724            // check for available blend operators
1725            {
1726                const bool can_blend_op =
1727                    0 != (m_d3dCaps.PrimitiveMiscCaps & D3DPMISCCAPS_BLENDOP);
1728                rt::enable_menu(menu, IDM_COLOR_BLEND_OP_SUBTRACT, can_blend_op);
1729                rt::enable_menu(menu, IDM_COLOR_BLEND_OP_REVERSE_SUBTRACT, can_blend_op);
1730                rt::enable_menu(menu, IDM_COLOR_BLEND_OP_MINIMUM, can_blend_op);
1731                rt::enable_menu(menu, IDM_COLOR_BLEND_OP_MAXIMUM, can_blend_op);
1732                if (!can_blend_op)
1733                {
1734                    if (m_color_blend_op != D3DBLENDOP_ADD)
1735                    {
1736                        m_color_blend_op = D3DBLENDOP_ADD;
```

```
1737                }
1738                if (m_alpha_blend_op != D3DBLENDOP_ADD)
1739                {
1740                    m_alpha_blend_op = D3DBLENDOP_ADD;
1741                }
1742            }
1743        }
1744
1745        // check source and destination blend factors
1746        const DWORD blend_caps[] =
1747        {
1748            D3DPBLENDCAPS_ZERO,
1749            D3DPBLENDCAPS_ONE,
1750            D3DPBLENDCAPS_SRCCOLOR,
1751            D3DPBLENDCAPS_INVSRCCOLOR,
1752            D3DPBLENDCAPS_SRCALPHA,
1753            D3DPBLENDCAPS_INVSRCALPHA,
1754            D3DPBLENDCAPS_DESTALPHA,
1755            D3DPBLENDCAPS_INVDESTALPHA,
1756            D3DPBLENDCAPS_DESTCOLOR,
1757            D3DPBLENDCAPS_INVDESTCOLOR,
1758            D3DPBLENDCAPS_SRCALPHASAT,
1759            0,
1760            D3DPBLENDCAPS_BLENDFACTOR,
1761            D3DPBLENDCAPS_BLENDFACTOR
1762        };
1763        UINT i;
1764        for (i = 0; i < NUM_OF(blend_caps); i++)
1765        {
1766            if (!blend_caps[i])
1767            {
1768                // D3DPBLENDCAPS_BOTHSRCALPHA is obsolete, so skip it.
1769                continue;
1770            }
1771
1772            bool support = 0 != (m_d3dCaps.SrcBlendCaps & blend_caps[i]);
1773            if (!support)
1774            {
1775                if (D3DBLEND(i+1) == m_color_src_blend)
1776                {
1777                    m_color_src_blend = D3DBLEND_ONE;
1778                }
1779                if (D3DBLEND(i+1) == m_alpha_src_blend)
1780                {
1781                    m_alpha_src_blend = D3DBLEND_ONE;
1782                }
```

```
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                    {
1792                        m_color_dest_blend = D3DBLEND_ZERO;
1793                    }
1794                    if (D3DBLEND(i+1) == m_alpha_dest_blend)
1795                    {
1796                        m_alpha_dest_blend = D3DBLEND_ZERO;
1797                    }
1798                }
1799                if (D3DBLEND_BOTHINVSRCALPHA != D3DBLEND(i+1))
1800                {
1801                    rt::enable_menu(menu, color_dest_blend_id(D3DBLEND(i+1)), support);
1802                    rt::enable_menu(menu, alpha_dest_blend_id(D3DBLEND(i+1)), support);
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);
```

```
1829                rt::enable_menu(menu, alpha_composite_id(e_composite_operator(i)),
1830                    support);
1831            }
1832
1833            // toggle menu items based on destination alpha supported
1834            if (!m_destination_alpha)
1835            {
1836                if (!m_color_allow_dest_alpha)
1837                {
1838                    const DWORD ids[] =
1839                    {
1840                        IDM_COLOR_COMPOSITE_DEST,
1841                        IDM_COLOR_COMPOSITE_DEST_OVER_SRC,
1842                        IDM_COLOR_COMPOSITE_SRC_IN_DEST,
1843                        IDM_COLOR_COMPOSITE_SRC_ATOP_DEST,
1844                        IDM_COLOR_COMPOSITE_DEST_ATOP_SRC,
1845                        IDM_COLOR_COMPOSITE_SRC_OUT_DEST,
1846                        IDM_COLOR_SRC_BLEND_DEST_ALPHA,
1847                        IDM_COLOR_SRC_BLEND_INV_DEST_ALPHA,
1848                        IDM_COLOR_SRC_BLEND_DEST_COLOR,
1849                        IDM_COLOR_SRC_BLEND_INV_DEST_COLOR,
1850                        IDM_COLOR_DEST_BLEND_DEST_ALPHA,
1851                        IDM_COLOR_DEST_BLEND_INV_DEST_ALPHA,
1852                        IDM_COLOR_DEST_BLEND_DEST_COLOR,
1853                        IDM_COLOR_DEST_BLEND_INV_DEST_COLOR
1854                    };
1855                    for (UINT i = 0; i < NUM_OF(ids); i++)
1856                    {
1857                        rt::enable_menu(menu, ids[i], false);
1858                    }
1859                }
1860                if (!m_alpha_allow_dest_alpha)
1861                {
1862                    const DWORD ids[] =
1863                    {
1864                        IDM_ALPHA_COMPOSITE_DEST,
1865                        IDM_ALPHA_COMPOSITE_DEST_OVER_SRC,
1866                        IDM_ALPHA_COMPOSITE_SRC_IN_DEST,
1867                        IDM_ALPHA_COMPOSITE_SRC_ATOP_DEST,
1868                        IDM_ALPHA_COMPOSITE_DEST_ATOP_SRC,
1869                        IDM_ALPHA_COMPOSITE_SRC_OUT_DEST,
1870                        IDM_ALPHA_SRC_BLEND_DEST_ALPHA,
1871                        IDM_ALPHA_SRC_BLEND_INV_DEST_ALPHA,
1872                        IDM_ALPHA_SRC_BLEND_DEST_COLOR,
1873                        IDM_ALPHA_SRC_BLEND_INV_DEST_COLOR,
1874                        IDM_ALPHA_DEST_BLEND_DEST_ALPHA,
```

```
1875                        IDM_ALPHA_DEST_BLEND_INV_DEST_ALPHA,
1876                        IDM_ALPHA_DEST_BLEND_DEST_COLOR,
1877                        IDM_ALPHA_DEST_BLEND_INV_DEST_COLOR
1878                    };
1879                    for (UINT i = 0; i < NUM_OF(ids); i++)
1880                    {
1881                        rt::enable_menu(menu, ids[i], false);
1882                    }
1883                }
1884            }
1885
1886            // the individual source and destination blend factors are not
1887            // used with min and max blend operators
1888            if ((D3DBLENDOP_MIN == m_color_blend_op) ||
1889                (D3DBLENDOP_MAX == m_color_blend_op))
1890            {
1891                DWORD ids[] =
1892                {
1893                    IDM_COLOR_COMPOSITE_CLEAR,
1894                    IDM_COLOR_COMPOSITE_SRC,
1895                    IDM_COLOR_COMPOSITE_DEST,
1896                    IDM_COLOR_COMPOSITE_SRC_OVER_DEST,
1897                    IDM_COLOR_COMPOSITE_DEST_OVER_SRC,
1898                    IDM_COLOR_COMPOSITE_SRC_IN_DEST,
1899                    IDM_COLOR_COMPOSITE_DEST_IN_SRC,
1900                    IDM_COLOR_COMPOSITE_SRC_ATOP_DEST,
1901                    IDM_COLOR_COMPOSITE_DEST_ATOP_SRC,
1902                    IDM_COLOR_COMPOSITE_SRC_OUT_DEST,
1903                    IDM_COLOR_COMPOSITE_DEST_OUT_SRC,
1904                    IDM_COLOR_COMPOSITE_SRC_XOR_DEST,
1905                    IDM_COLOR_SRC_BLEND_ZERO,
1906                    IDM_COLOR_SRC_BLEND_SRC_COLOR,
1907                    IDM_COLOR_SRC_BLEND_INV_SRC_COLOR,
1908                    IDM_COLOR_SRC_BLEND_SRC_ALPHA,
1909                    IDM_COLOR_SRC_BLEND_INV_SRC_ALPHA,
1910                    IDM_COLOR_SRC_BLEND_DEST_ALPHA,
1911                    IDM_COLOR_SRC_BLEND_INV_DEST_ALPHA,
1912                    IDM_COLOR_SRC_BLEND_DEST_COLOR,
1913                    IDM_COLOR_SRC_BLEND_INV_DEST_COLOR,
1914                    IDM_COLOR_SRC_BLEND_SRC_ALPHA_SAT,
1915                    IDM_COLOR_SRC_BLEND_BOTH_INV_SRC_ALPHA,
1916                    IDM_COLOR_DEST_BLEND_ONE,
1917                    IDM_COLOR_DEST_BLEND_SRC_COLOR,
1918                    IDM_COLOR_DEST_BLEND_INV_SRC_COLOR,
1919                    IDM_COLOR_DEST_BLEND_SRC_ALPHA,
1920                    IDM_COLOR_DEST_BLEND_INV_SRC_ALPHA,
```

```
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_INV_SRC_COLOR,
1954                IDM_ALPHA_SRC_BLEND_SRC_ALPHA,
1955                IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA,
1956                IDM_ALPHA_SRC_BLEND_DEST_ALPHA,
1957                IDM_ALPHA_SRC_BLEND_INV_DEST_ALPHA,
1958                IDM_ALPHA_SRC_BLEND_DEST_COLOR,
1959                IDM_ALPHA_SRC_BLEND_INV_DEST_COLOR,
1960                IDM_ALPHA_SRC_BLEND_SRC_ALPHA_SAT,
1961                IDM_ALPHA_SRC_BLEND_BOTH_INV_SRC_ALPHA,
1962                IDM_ALPHA_DEST_BLEND_ONE,
1963                IDM_ALPHA_DEST_BLEND_SRC_COLOR,
1964                IDM_ALPHA_DEST_BLEND_INV_SRC_COLOR,
1965                IDM_ALPHA_DEST_BLEND_SRC_ALPHA,
1966                IDM_ALPHA_DEST_BLEND_INV_SRC_ALPHA,
```

```
1967                    IDM_ALPHA_DEST_BLEND_DEST_ALPHA,
1968                    IDM_ALPHA_DEST_BLEND_INV_DEST_ALPHA,
1969                    IDM_ALPHA_DEST_BLEND_DEST_COLOR,
1970                    IDM_ALPHA_DEST_BLEND_INV_DEST_COLOR
1971                };
1972                for (i = 0; i < NUM_OF(ids); i++)
1973                {
1974                    rt::enable_menu(menu, ids[i], false);
1975                }
1976                m_alpha_composite = E_COMPOSITE_NONE;
1977                m_alpha_src_blend = D3DBLEND_ONE;
1978                m_alpha_dest_blend = D3DBLEND_ZERO;
1979            }
1980
1981            // now patch up m_color_composite in case it is inconsistent with
1982            // m_color_src_blend and m_color_dest_blend
1983            for (i = 0; i < NUM_OF(sm_blend_factors); i++)
1984            {
1985                if ((sm_blend_factors[i].m_src == m_color_src_blend) &&
1986                    (sm_blend_factors[i].m_dest == m_color_dest_blend))
1987                {
1988                    m_color_composite = e_composite_operator(i);
1989                    break;
1990                }
1991            }
1992            if (NUM_OF(sm_blend_factors) == i)
1993            {
1994                m_color_composite = E_COMPOSITE_NONE;
1995            }
1996            for (i = 0; i < NUM_OF(sm_blend_factors); i++)
1997            {
1998                if ((sm_blend_factors[i].m_src == m_alpha_src_blend) &&
1999                    (sm_blend_factors[i].m_dest == m_alpha_dest_blend))
2000                {
2001                    m_alpha_composite = e_composite_operator(i);
2002                    break;
2003                }
2004            }
2005            if (NUM_OF(sm_blend_factors) == i)
2006            {
2007                m_alpha_composite = E_COMPOSITE_NONE;
2008            }
2009        }
2010
2011        rt::check_menu(menu, IDM_COLOR_BLEND_ENABLE, m_color_blend_enable);
2012        rt::check_menu(menu, IDM_ALPHA_BLEND_ENABLE, m_alpha_blend_enable);
```

```
2013        rt::check_menu(menu, color_blend_op_id(m_color_blend_op), checked);
2014        rt::check_menu(menu, alpha_blend_op_id(m_alpha_blend_op), checked);
2015        if (E_COMPOSITE_NONE != m_color_composite)
2016        {
2017            rt::check_menu(menu, color_composite_id(m_color_composite), checked);
2018        }
2019        rt::check_menu(menu, color_src_blend_id(m_color_src_blend), checked);
2020        rt::check_menu(menu, color_dest_blend_id(m_color_dest_blend), checked);
2021        if (E_COMPOSITE_NONE != m_alpha_composite)
2022        {
2023            rt::check_menu(menu, alpha_composite_id(m_alpha_composite), checked);
2024        }
2025        rt::check_menu(menu, alpha_src_blend_id(m_alpha_src_blend), checked);
2026        rt::check_menu(menu, alpha_dest_blend_id(m_alpha_dest_blend), checked);
2027
2028        // disable blending submenus if blending is disabled
2029        rt::enable_menu(menu, IDM_COLOR_BLEND_DEST_ALPHA, m_color_blend_enable);
2030        enable_menu(menu, _T("&Alpha Blend"), m_color_blend_enable);
2031        TWS(::DrawMenuBar(m_hWnd));
2032        HMENU blend_menu = TWS(find_menu(menu, _T("&Color Blend")));
2033        enable_menu(blend_menu, _T("&Operator"), m_color_blend_enable);
2034        enable_menu(blend_menu, _T("&Compositing"), m_color_blend_enable);
2035        enable_menu(blend_menu, _T("&Source"), m_color_blend_enable);
2036        enable_menu(blend_menu, _T("&Destination"), m_color_blend_enable);
2037
2038        rt::enable_menu(menu, IDM_ALPHA_BLEND_DEST_ALPHA, m_alpha_blend_enable);
2039        blend_menu = TWS(find_menu(menu, _T("&Alpha Blend")));
2040        enable_menu(blend_menu, _T("&Operator"), m_alpha_blend_enable);
2041        enable_menu(blend_menu, _T("&Compositing"), m_alpha_blend_enable);
2042        enable_menu(blend_menu, _T("&Source"), m_alpha_blend_enable);
2043        enable_menu(blend_menu, _T("&Destination"), m_alpha_blend_enable);
2044    }
2045
2046    //////////////////////////////////////////////////////////////////////
2047    // CMyD3DApplication::on_command
2048    //
2049    // WM_COMMAND handler for this sample.  This mostly amounts to updating
2050    // internal variables that reflect the current state of the UI, although
2051    // some D3D state is updated directly from here.
2052    //
2053    LRESULT
2054    CMyD3DApplication::on_command(HWND window, WPARAM wp, LPARAM, bool &handled)
2055    {
2056        const HMENU menu = ::GetMenu(window);
2057        const UINT control = LOWORD(wp);
2058
```

```
2059        switch (control)
2060        {
2061    #define TOGGLE_COLOR_WRITE_ENABLE(id_, state_)                      \
2062        case IDM_COLOR_WRITE_ENABLE_##id_:                              \
2063            rt::toggle_menu(menu, IDM_COLOR_WRITE_ENABLE_##id_, state_); \
2064            set_color_write_enable();                                   \
2065            handled = true;                                             \
2066            break
2067        TOGGLE_COLOR_WRITE_ENABLE(RED, m_color_write_enable_red);
2068        TOGGLE_COLOR_WRITE_ENABLE(GREEN, m_color_write_enable_green);
2069        TOGGLE_COLOR_WRITE_ENABLE(BLUE, m_color_write_enable_blue);
2070        TOGGLE_COLOR_WRITE_ENABLE(ALPHA, m_color_write_enable_alpha);
2071    #undef TOGGLE_COLOR_WRITE_ENABLE
2072    #define TOGGLE(id_, state_)                 \
2073        case id_:                               \
2074            rt::toggle_menu(menu, id_, state_); \
2075            handled = true;                     \
2076            break
2077        TOGGLE(IDM_Z_WRITE_ENABLE, m_z_write_enable);
2078        TOGGLE(IDM_ANIMATE_VIEW, m_animate_view);
2079        TOGGLE(IDM_SHOW_STATS, m_show_stats);
2080        TOGGLE(IDM_OPTION_DITHER, m_dithered);
2081        TOGGLE(IDM_OPTION_BACKGROUND_TILE, m_tile_background);
2082        TOGGLE(ID_OPTIONS_SCISSORTEST, m_scissor_enable);
2083    #undef TOGGLE
2084
2085    #define MODIFY_COLOR(id_, state_, transparent_) \
2086        case id_: \
2087            { \
2088                rt::pauser block(*this); \
2089                state_ = rt::choose_color_transparent(window, state_, transparent_); \
2090            } \
2091            handled = true; \
2092            break
2093        MODIFY_COLOR(IDM_EDIT_DIFFUSE_COLOR, m_fg, true);
2094        MODIFY_COLOR(IDM_EDIT_TEXT_COLOR, m_text_fg, true);
2095        MODIFY_COLOR(IDM_EDIT_SPECULAR_COLOR, m_specular, false);
2096    #undef MODIFY_COLOR
2097
2098        case IDM_EDIT_BLEND_FACTOR:
2099            {
2100                rt::pauser block(*this);
2101                m_blend_factor = rt::choose_color(window, m_blend_factor);
2102            }
2103            handled = true;
2104            break;
```

```
2105
2106        case IDM_EDIT_BG_COLOR:
2107            {
2108                rt::pauser block(*this);
2109                m_bg = rt::choose_color(window, m_bg);
2110            }
2111            handled = true;
2112            break;
2113
2114        case IDM_TEXTURED:
2115            m_textured = m_can_texture && !m_textured;
2116            rt::check_menu(menu, IDM_TEXTURED, m_textured);
2117            handled = true;
2118            break;
2119
2120        case IDM_FILE_OPEN_TEXTURE:
2121            {
2122                rt::pauser block(*this);
2123                on_texture_file();
2124            }
2125            handled = true;
2126            break;
2127
2128        case IDM_STENCIL_IRREGULAR_MASK:
2129            rt::toggle_menu(menu, IDM_STENCIL_IRREGULAR_MASK, m_stencil_mask);
2130            m_build_stencil = true;
2131            handled = true;
2132            break;
2133
2134        case IDM_STENCIL_STIPPLING:
2135            rt::toggle_menu(menu, IDM_STENCIL_STIPPLING, m_stencil_stippling);
2136            m_build_stencil = true;
2137            handled = true;
2138            break;
2139
2140        case IDM_MULTISAMPLE_ANTIALIAS:
2141            rt::toggle_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, m_multisample_antialias
2142            m_multisample_depth_of_field = false;
2143            m_multisample_motion_blur = false;
2144            rt::check_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, m_multisample_depth
2145            rt::check_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, m_multisample_motion_b
2146            handled = true;
2147            break;
2148
2149        case IDM_MULTISAMPLE_DEPTH_OF_FIELD:
2150            rt::toggle_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, m_multisample_dept
```

```
2151            m_multisample_antialias = false;
2152            m_multisample_motion_blur = false;
2153            rt::check_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, m_multisample_antialias);
2154            rt::check_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, m_multisample_motion_blur);
2155            handled = true;
2156            break;
2157
2158        case IDM_MULTISAMPLE_MOTION_BLUR:
2159            rt::toggle_menu(menu, IDM_MULTISAMPLE_MOTION_BLUR, m_multisample_motion_blur);
2160            m_multisample_antialias = false;
2161            m_multisample_depth_of_field = false;
2162            rt::check_menu(menu, IDM_MULTISAMPLE_ANTIALIAS, m_multisample_antialias);
2163            rt::check_menu(menu, IDM_MULTISAMPLE_DEPTH_OF_FIELD, m_multisample_depth_of_field)
2164            handled = true;
2165            break;
2166
2167        case IDM_COLOR_SRC_BLEND_ZERO:
2168        case IDM_COLOR_SRC_BLEND_ONE:
2169        case IDM_COLOR_SRC_BLEND_SRC_COLOR:
2170        case IDM_COLOR_SRC_BLEND_INV_SRC_COLOR:
2171        case IDM_COLOR_SRC_BLEND_SRC_ALPHA:
2172        case IDM_COLOR_SRC_BLEND_INV_SRC_ALPHA:
2173        case IDM_COLOR_SRC_BLEND_DEST_ALPHA:
2174        case IDM_COLOR_SRC_BLEND_INV_DEST_ALPHA:
2175        case IDM_COLOR_SRC_BLEND_DEST_COLOR:
2176        case IDM_COLOR_SRC_BLEND_INV_DEST_COLOR:
2177        case IDM_COLOR_SRC_BLEND_SRC_ALPHA_SAT:
2178        case IDM_COLOR_SRC_BLEND_BOTH_INV_SRC_ALPHA:
2179        case IDM_COLOR_SRC_BLEND_FACTOR:
2180        case IDM_COLOR_SRC_BLEND_INV_FACTOR:
2181            update_blending(menu, false);
2182            m_color_src_blend = D3DBLEND(get_menu_data(control));
2183            update_blending(menu, true);
2184            handled = true;
2185            break;
2186
2187        case IDM_COLOR_DEST_BLEND_ZERO:
2188        case IDM_COLOR_DEST_BLEND_ONE:
2189        case IDM_COLOR_DEST_BLEND_SRC_COLOR:
2190        case IDM_COLOR_DEST_BLEND_INV_SRC_COLOR:
2191        case IDM_COLOR_DEST_BLEND_SRC_ALPHA:
2192        case IDM_COLOR_DEST_BLEND_INV_SRC_ALPHA:
2193        case IDM_COLOR_DEST_BLEND_DEST_ALPHA:
2194        case IDM_COLOR_DEST_BLEND_INV_DEST_ALPHA:
2195        case IDM_COLOR_DEST_BLEND_DEST_COLOR:
2196        case IDM_COLOR_DEST_BLEND_INV_DEST_COLOR:
```

```
2197        case IDM_COLOR_DEST_BLEND_SRC_ALPHA_SAT:
2198        case IDM_COLOR_DEST_BLEND_FACTOR:
2199        case IDM_COLOR_DEST_BLEND_INV_FACTOR:
2200            update_blending(menu, false);
2201            m_color_dest_blend = D3DBLEND(get_menu_data(control));
2202            update_blending(menu, true);
2203            handled = true;
2204            break;
2205
2206        case IDM_COLOR_COMPOSITE_CLEAR:
2207        case IDM_COLOR_COMPOSITE_SRC:
2208        case IDM_COLOR_COMPOSITE_DEST:
2209        case IDM_COLOR_COMPOSITE_SRC_OVER_DEST:
2210        case IDM_COLOR_COMPOSITE_DEST_OVER_SRC:
2211        case IDM_COLOR_COMPOSITE_SRC_IN_DEST:
2212        case IDM_COLOR_COMPOSITE_DEST_IN_SRC:
2213        case IDM_COLOR_COMPOSITE_SRC_ATOP_DEST:
2214        case IDM_COLOR_COMPOSITE_DEST_ATOP_SRC:
2215        case IDM_COLOR_COMPOSITE_SRC_OUT_DEST:
2216        case IDM_COLOR_COMPOSITE_DEST_OUT_SRC:
2217        case IDM_COLOR_COMPOSITE_SRC_XOR_DEST:
2218            update_blending(menu, false);
2219            m_color_composite = e_composite_operator(get_menu_data(control));
2220            m_color_src_blend = sm_blend_factors[m_color_composite].m_src;
2221            m_color_dest_blend = sm_blend_factors[m_color_composite].m_dest;
2222            update_blending(menu, true);
2223            handled = true;
2224            break;
2225
2226        case IDM_COLOR_BLEND_DEST_ALPHA:
2227            update_blending(menu, false);
2228            rt::toggle_menu(menu, IDM_COLOR_BLEND_DEST_ALPHA, m_color_allow_dest_alp
2229            update_blending(menu, true);
2230            handled = true;
2231            break;
2232
2233        case IDM_COLOR_BLEND_ENABLE:
2234            update_blending(menu, false);
2235            m_color_blend_enable = !m_color_blend_enable;
2236            update_blending(menu, true);
2237            handled = true;
2238            break;
2239
2240        case IDM_COLOR_BLEND_OP_ADD:
2241        case IDM_COLOR_BLEND_OP_SUBTRACT:
2242        case IDM_COLOR_BLEND_OP_REVERSE_SUBTRACT:
```

```
2243        case IDM_COLOR_BLEND_OP_MINIMUM:
2244        case IDM_COLOR_BLEND_OP_MAXIMUM:
2245            update_blending(menu, false);
2246            m_color_blend_op = D3DBLENDOP(get_menu_data(control));
2247            update_blending(menu, true);
2248            handled = true;
2249            break;
2250
2251        case IDM_ALPHA_SRC_BLEND_ZERO:
2252        case IDM_ALPHA_SRC_BLEND_ONE:
2253        case IDM_ALPHA_SRC_BLEND_SRC_COLOR:
2254        case IDM_ALPHA_SRC_BLEND_INV_SRC_COLOR:
2255        case IDM_ALPHA_SRC_BLEND_SRC_ALPHA:
2256        case IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA:
2257        case IDM_ALPHA_SRC_BLEND_DEST_ALPHA:
2258        case IDM_ALPHA_SRC_BLEND_INV_DEST_ALPHA:
2259        case IDM_ALPHA_SRC_BLEND_DEST_COLOR:
2260        case IDM_ALPHA_SRC_BLEND_INV_DEST_COLOR:
2261        case IDM_ALPHA_SRC_BLEND_SRC_ALPHA_SAT:
2262        case IDM_ALPHA_SRC_BLEND_BOTH_INV_SRC_ALPHA:
2263        case IDM_ALPHA_SRC_BLEND_FACTOR:
2264        case IDM_ALPHA_SRC_BLEND_INV_FACTOR:
2265            update_blending(menu, false);
2266            m_alpha_src_blend = D3DBLEND(get_menu_data(control));
2267            update_blending(menu, true);
2268            handled = true;
2269            break;
2270
2271        case IDM_ALPHA_DEST_BLEND_ZERO:
2272        case IDM_ALPHA_DEST_BLEND_ONE:
2273        case IDM_ALPHA_DEST_BLEND_SRC_COLOR:
2274        case IDM_ALPHA_DEST_BLEND_INV_SRC_COLOR:
2275        case IDM_ALPHA_DEST_BLEND_SRC_ALPHA:
2276        case IDM_ALPHA_DEST_BLEND_INV_SRC_ALPHA:
2277        case IDM_ALPHA_DEST_BLEND_DEST_ALPHA:
2278        case IDM_ALPHA_DEST_BLEND_INV_DEST_ALPHA:
2279        case IDM_ALPHA_DEST_BLEND_DEST_COLOR:
2280        case IDM_ALPHA_DEST_BLEND_INV_DEST_COLOR:
2281        case IDM_ALPHA_DEST_BLEND_SRC_ALPHA_SAT:
2282        case IDM_ALPHA_DEST_BLEND_FACTOR:
2283        case IDM_ALPHA_DEST_BLEND_INV_FACTOR:
2284            update_blending(menu, false);
2285            m_alpha_dest_blend = D3DBLEND(get_menu_data(control));
2286            update_blending(menu, true);
2287            handled = true;
2288            break;
```

```
2289
2290        case IDM_ALPHA_COMPOSITE_CLEAR:
2291        case IDM_ALPHA_COMPOSITE_SRC:
2292        case IDM_ALPHA_COMPOSITE_DEST:
2293        case IDM_ALPHA_COMPOSITE_SRC_OVER_DEST:
2294        case IDM_ALPHA_COMPOSITE_DEST_OVER_SRC:
2295        case IDM_ALPHA_COMPOSITE_SRC_IN_DEST:
2296        case IDM_ALPHA_COMPOSITE_DEST_IN_SRC:
2297        case IDM_ALPHA_COMPOSITE_SRC_ATOP_DEST:
2298        case IDM_ALPHA_COMPOSITE_DEST_ATOP_SRC:
2299        case IDM_ALPHA_COMPOSITE_SRC_OUT_DEST:
2300        case IDM_ALPHA_COMPOSITE_DEST_OUT_SRC:
2301        case IDM_ALPHA_COMPOSITE_SRC_XOR_DEST:
2302            update_blending(menu, false);
2303            m_alpha_composite = e_composite_operator(get_menu_data(control));
2304            m_alpha_src_blend = sm_blend_factors[m_alpha_composite].m_src;
2305            m_alpha_dest_blend = sm_blend_factors[m_alpha_composite].m_dest;
2306            update_blending(menu, true);
2307            handled = true;
2308            break;
2309
2310        case IDM_ALPHA_BLEND_DEST_ALPHA:
2311            update_blending(menu, false);
2312            rt::toggle_menu(menu, IDM_ALPHA_BLEND_DEST_ALPHA, m_alpha_allow_dest_alp
2313            update_blending(menu, true);
2314            handled = true;
2315            break;
2316
2317        case IDM_ALPHA_BLEND_ENABLE:
2318            update_blending(menu, false);
2319            m_alpha_blend_enable = !m_alpha_blend_enable;
2320            update_blending(menu, true);
2321            handled = true;
2322            break;
2323
2324        case IDM_ALPHA_BLEND_OP_ADD:
2325        case IDM_ALPHA_BLEND_OP_SUBTRACT:
2326        case IDM_ALPHA_BLEND_OP_REVERSE_SUBTRACT:
2327        case IDM_ALPHA_BLEND_OP_MINIMUM:
2328        case IDM_ALPHA_BLEND_OP_MAXIMUM:
2329            update_blending(menu, false);
2330            m_alpha_blend_op = D3DBLENDOP(get_menu_data(control));
2331            update_blending(menu, true);
2332            handled = true;
2333            break;
2334
```

```
2335        case IDM_RESET_VIEW:
2336            m_rot_x = 0.0f;
2337            m_rot_y = 0.0f;
2338            handled = true;
2339            break;
2340
2341        default:
2342            // all our control IDs are > 40006 and we should handle them all
2343            if (control > 40006)
2344            {
2345                ATLASSERT(false);
2346            }
2347        }
2348
2349        return 0;
2350    }
2351
2352    //-----------------------------------------------------------------------------
2353    // CMyD3DApplication::on_texture_file
2354    //
2355    // Get a texture filename from the user and try to open it as the current
2356    // texture.
2357    //
2358    void
2359    CMyD3DApplication::on_texture_file()
2360    {
2361        TCHAR file[MAX_PATH] = { 0 };
2362        OPENFILENAME ofn =
2363        {
2364            sizeof(ofn), 0, 0,
2365                _T("All images\0")
2366                    _T("*.bmp;*.jpg;*.png;*.pbm;*.pgm;*.ppm;*.pnm;*.tga;*.tif\0")
2367                _T("All files (*.*)\0*.*\0")
2368                _T("Bitmap images (*.bmp)\0*.bmp\0")
2369                _T("JPEG images (*.jpg)\0*.jpg\0")
2370                _T("PNG images (*.png)\0*.png\0")
2371                _T("Portable bitmap (*.pbm)\0*.pbm\0")
2372                _T("Portable graymap (*.pgm)\0*.pgm\0")
2373                _T("Portable pixmap (*.ppm)\0*.ppm\0")
2374                _T("Portable anymap (*.pnm)\0*.pnm\0")
2375                _T("Targa images(*.tga)\0*.tga\0")
2376                _T("TIFF images (*.tif)\0*.tif\0")
2377                _T("\0"),
2378            0,  0, 0,
2379            file, MAX_PATH, 0,  0, 0, 0, OFN_FILEMUSTEXIST
2380        };
```

```
2381        if (::GetOpenFileName(&ofn))
2382        {
2383            create_texture(file);
2384        }
2385    }
2386
2387    ////////////////////////////////////////////////////////
2388    // set_color_write_enable
2389    //
2390    // Set RS Color Write Enable based on UI state.
2391    //
2392    void
2393    CMyD3DApplication::set_color_write_enable()
2394    {
2395        THR(m_pd3dDevice->SetRenderState(D3DRS_COLORWRITEENABLE,
2396            (m_color_write_enable_red ? D3DCOLORWRITEENABLE_RED : 0) |
2397            (m_color_write_enable_green ? D3DCOLORWRITEENABLE_GREEN : 0) |
2398            (m_color_write_enable_blue ? D3DCOLORWRITEENABLE_BLUE : 0) |
2399            (m_color_write_enable_alpha ? D3DCOLORWRITEENABLE_ALPHA : 0)));
2400    }
2401
2402    ////////////////////////////////////////////////////////
2403    // render_stencil_mask
2404    //
2405    // Use the torus mesh to render a hollow circle mask into
2406    // the 0x1 stencil plane.
2407    //
2408    void
2409    CMyD3DApplication::render_stencil_mask()
2410    {
2411        // set the reference and mask values for plane 1
2412        rt::s_rs states[] =
2413        {
2414            D3DRS_STENCILREF, 0x1,
2415            D3DRS_STENCILMASK, 0x1,
2416            D3DRS_STENCILWRITEMASK, 0x1,
2417        };
2418        rt::set_states(m_pd3dDevice, states, NUM_OF(states));
2419        // disable texturing
2420        const rt::s_tss ts_states[] =
2421        {
2422            D3DTSS_COLOROP, D3DTOP_DISABLE,
2423            D3DTSS_ALPHAOP, D3DTOP_DISABLE
2424        };
2425        rt::set_states(m_pd3dDevice, 0, ts_states,
2426            NUM_OF(ts_states));
```

```
2427        THR(m_pd3dDevice->SetTexture(0, NULL));
2428
2429        // draw torus
2430        D3DXMATRIX ident(1, 0, 0, 0,
2431                         0, 1, 0, 0,
2432                         0, 0, 1, 0,
2433                         0, 0, 0, 1);
2434        THR(m_pd3dDevice->SetTransform(D3DTS_WORLD, &ident));
2435        THR(m_torus->DrawSubset(0));
2436    }
2437
2438    ////////////////////////////////////////////////////////
2439    // render_stencil_stipple
2440    //
2441    // Render a stipple pattern into the 0x2 stencil plane.
2442    //
2443    void
2444    CMyD3DApplication::render_stencil_stipple()
2445    {
2446        // set the reference and mask values for plane 2; the
2447        // alpha test rejects stipple texels with zero alpha,
2448        // so that the stencil bits are not written there.
2449        rt::s_rs states[] =
2450        {
2451            D3DRS_ALPHATESTENABLE, true,
2452            D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL,
2453            D3DRS_ALPHAREF, 0x1,
2454            D3DRS_STENCILREF, 0x2,
2455            D3DRS_STENCILMASK, 0x2,
2456            D3DRS_STENCILWRITEMASK, 0x2,
2457            D3DRS_COLORVERTEX, true,
2458            D3DRS_LIGHTING, false,
2459            D3DRS_SPECULARENABLE, false
2460        };
2461        rt::set_states(m_pd3dDevice, states, NUM_OF(states));
2462        rt::s_tss tex_states[] =
2463        {
2464            D3DTSS_COLOROP, D3DTOP_SELECTARG1,
2465            D3DTSS_COLORARG1, D3DTA_TEXTURE,
2466            D3DTSS_COLORARG2, D3DTA_DIFFUSE,
2467            D3DTSS_ALPHAOP, D3DTOP_SELECTARG1,
2468            D3DTSS_ALPHAARG1, D3DTA_TEXTURE,
2469            D3DTSS_ALPHAARG2, D3DTA_DIFFUSE
2470        };
2471        rt::set_states(m_pd3dDevice, 0,
2472            tex_states, NUM_OF(tex_states));
```

```
2473        THR(m_pd3dDevice->SetTexture(0, m_stipple));
2474
2475        // draw stipple with screen-space triangles that cover
2476        // the whole screen
2477        THR(m_pd3dDevice->SetFVF(s_screen_vertex::FVF));
2478        THR(m_pd3dDevice->SetStreamSource(0, m_stipple_verts, 0,
2479            sizeof(s_screen_vertex)));
2480        THR(m_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0,
2481            m_num_stipple_quads*2));
2482    }
2483
2484    ////////////////////////////////////////////////////////
2485    // render_stencil
2486    //
2487    // Build the stencil plane(s) depending on UI state.
2488    //
2489    void
2490    CMyD3DApplication::render_stencil()
2491    {
2492        // clear the stencil planes first
2493        THR(m_pd3dDevice->Clear(0L, NULL, D3DCLEAR_STENCIL,
2494            0, 1.0f, 0L));
2495
2496        // set states for stencil rendering
2497        rt::s_rs states[] =
2498        {
2499            D3DRS_STENCILENABLE, true,
2500            D3DRS_STENCILFUNC, D3DCMP_ALWAYS,
2501            D3DRS_STENCILPASS, D3DSTENCILOP_REPLACE,
2502            D3DRS_STENCILFAIL, D3DSTENCILOP_REPLACE,
2503            D3DRS_STENCILZFAIL, D3DSTENCILOP_REPLACE,
2504            D3DRS_COLORWRITEENABLE, 0L,
2505            D3DRS_ALPHABLENDENABLE, m_color_blend_enable,
2506            D3DRS_SRCBLEND, D3DBLEND_ZERO,
2507            D3DRS_DESTBLEND, D3DBLEND_ONE
2508        };
2509        rt::set_states(m_pd3dDevice, states, NUM_OF(states));
2510
2511        // render geometry to set the planes
2512        if (m_stencil_mask)
2513        {
2514            render_stencil_mask();
2515        }
2516        if (m_stencil_stippling)
2517        {
2518            render_stencil_stipple();
```

```
2519        }
2520
2521        // set the state for masking against stencil bits
2522        rt::s_rs use_stencil_states[] =
2523        {
2524            D3DRS_STENCILENABLE, true,
2525            D3DRS_STENCILFUNC, D3DCMP_EQUAL,
2526            D3DRS_STENCILREF, 0,
2527            D3DRS_STENCILMASK, ~0UL,
2528            D3DRS_STENCILWRITEMASK, 0,
2529            D3DRS_STENCILPASS, D3DSTENCILOP_KEEP,
2530            D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP,
2531            D3DRS_STENCILZFAIL, D3DSTENCILOP_KEEP
2532        };
2533        rt::set_states(m_pd3dDevice, use_stencil_states,
2534            NUM_OF(use_stencil_states));
2535        set_color_write_enable();
2536    }
2537
2538    ////////////////////////////////////////////////////////
2539    // render_teapot
2540    //
2541    // Render the teapot with a given world matrix.
2542    //
2543    void
2544    CMyD3DApplication::render_teapot(const D3DXMATRIX &matrix)
2545    {
2546        THR(m_pd3dDevice->SetTransform(D3DTS_WORLD, &matrix));
2547        THR(m_teapot->DrawSubset(0));
2548    }
2549
2550    ////////////////////////////////////////////////////////
2551    // render_motion_blur
2552    //
2553    // Render multiple copies of the teapot, changing the
2554    // multisample mask to create a motion blur effect.
2555    //
2556    void
2557    CMyD3DApplication::render_motion_blur()
2558    {
2559        UINT passes = std::max(UINT(m_d3dpp.MultiSampleType), 1U);
2560        float rot_x = m_last_rot_x;
2561        float delta_x = (m_rot_x - m_last_rot_x)/passes;
2562        float rot_y = m_last_rot_y;
2563        float delta_y = (m_rot_y - m_last_rot_y)/passes;
2564
```

```
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
2590        const UINT passes =
2591            std::max(UINT(m_d3dpp.MultiSampleType), 1U);
2592        const float *jitter = sm_jitter[passes-1];
2593        for (UINT i = 0; i < passes; i++)
2594        {
2595            THR(m_pd3dDevice->SetRenderState(
2596                D3DRS_MULTISAMPLEMASK, 1L << i));
2597
2598            rt::mat_look_at view(D3DXVECTOR3(
2599                jitter[i*2+0]*0.25f, jitter[i*2+1]*0.25f, -5));
2600            THR(m_pd3dDevice->SetTransform(D3DTS_VIEW, &view));
2601
2602            render_teapot(base*rt::mat_trans(-1, -1, 1));
2603            render_teapot(base*rt::mat_trans(1, -1, 1));
2604            render_teapot(base*rt::mat_trans(-1, 1, 1));
2605            render_teapot(base*rt::mat_trans(1, 1, 1));
2606            render_teapot(base);
2607            render_teapot(base*rt::mat_trans(-1, -1, -1));
2608            render_teapot(base*rt::mat_trans(1, -1, -1));
2609            render_teapot(base*rt::mat_trans(-1, 1, -1));
2610            render_teapot(base*rt::mat_trans(1, 1, -1));
```

```
2611        }
2612
2613        rt::mat_look_at view(D3DXVECTOR3(0, 0, -5));
2614        THR(m_pd3dDevice->SetTransform(D3DTS_VIEW, &view));
2615        THR(m_pd3dDevice->SetRenderState(D3DRS_MULTISAMPLEMASK,
2616            ~0UL));
2617    }
2618
2619    //////////////////////////////////////////////////////////////////////
2620    // CMyD3DApplication::tile_background
2621    //
2622    // Use CopyRects to tile the back buffer with an image containing alpha.
2623    // The rectangle and point lists are rebuilt whenever RestoreDeviceObjects
2624    // is called.  This keeps the lists updated whenever the device changes
2625    // or the window is resized.
2626    //
2627    void
2628    CMyD3DApplication::tile_background()
2629    {
2630        // draw background tile, storing alpha straight into destination
2631        CComPtr<IDirect3DSurface9> back;
2632        THR(m_pd3dDevice->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &back));
2633        for (UINT i = 0; i < m_tile_rects.size(); i++)
2634        {
2635            RECT r =
2636            {
2637                m_tile_offsets[i].x, m_tile_offsets[i].y,
2638                m_tile_offsets[i].x + m_tile_rects[i].right - m_tile_rects[i].left,
2639                m_tile_offsets[i].y + m_tile_rects[i].bottom - m_tile_rects[i].top
2640            };
2641            THR(m_pd3dDevice->StretchRect(m_device_tile, &m_tile_rects[i],
2642                back, &r, D3DTEXF_NONE));
2643        }
2644    }
2645
2646    DWORD
2647    CMyD3DApplication::PresentFlags() const
2648    {
2649        DWORD flags = CD3DApplication::PresentFlags();
2650        if (m_d3dEnumeration.AppMinStencilBits > 0)
2651        {
2652            flags &= ~D3DPRESENTFLAG_DISCARD_DEPTHSTENCIL;
2653        }
2654        return flags;
2655    }
2656
```

```
2657  struct s_menu_data
2658  {
2659      UINT m_id;
2660      DWORD m_data;
2661  };
2662
2663  void
2664  CMyD3DApplication::set_menu_data()
2665  {
2666      const s_menu_data data[] =
2667      {
2668          IDM_COLOR_COMPOSITE_CLEAR,              E_COMPOSITE_CLEAR,
2669          IDM_COLOR_COMPOSITE_SRC,                E_COMPOSITE_SRC,
2670          IDM_COLOR_COMPOSITE_DEST,               E_COMPOSITE_DEST,
2671          IDM_COLOR_COMPOSITE_SRC_OVER_DEST,      E_COMPOSITE_SRC_OVER_DEST,
2672          IDM_COLOR_COMPOSITE_DEST_OVER_SRC,      E_COMPOSITE_DEST_OVER_SRC,
2673          IDM_COLOR_COMPOSITE_SRC_IN_DEST,        E_COMPOSITE_SRC_IN_DEST,
2674          IDM_COLOR_COMPOSITE_DEST_IN_SRC,        E_COMPOSITE_DEST_IN_SRC,
2675          IDM_COLOR_COMPOSITE_SRC_ATOP_DEST,      E_COMPOSITE_SRC_ATOP_DEST,
2676          IDM_COLOR_COMPOSITE_DEST_ATOP_SRC,      E_COMPOSITE_DEST_ATOP_SRC,
2677          IDM_COLOR_COMPOSITE_SRC_OUT_DEST,       E_COMPOSITE_SRC_OUT_DEST,
2678          IDM_COLOR_COMPOSITE_DEST_OUT_SRC,       E_COMPOSITE_DEST_OUT_SRC,
2679          IDM_COLOR_COMPOSITE_SRC_XOR_DEST,       E_COMPOSITE_SRC_XOR_DEST,
2680          IDM_COLOR_BLEND_OP_ADD,                 D3DBLENDOP_ADD,
2681          IDM_COLOR_BLEND_OP_SUBTRACT,            D3DBLENDOP_SUBTRACT,
2682          IDM_COLOR_BLEND_OP_REVERSE_SUBTRACT,    D3DBLENDOP_REVSUBTRACT,
2683          IDM_COLOR_BLEND_OP_MINIMUM,             D3DBLENDOP_MIN,
2684          IDM_COLOR_BLEND_OP_MAXIMUM,             D3DBLENDOP_MAX,
2685          IDM_COLOR_SRC_BLEND_ZERO,               D3DBLEND_ZERO,
2686          IDM_COLOR_SRC_BLEND_ONE,                D3DBLEND_ONE,
2687          IDM_COLOR_SRC_BLEND_SRC_COLOR,          D3DBLEND_SRCCOLOR,
2688          IDM_COLOR_SRC_BLEND_INV_SRC_COLOR,      D3DBLEND_INVSRCCOLOR,
2689          IDM_COLOR_SRC_BLEND_SRC_ALPHA,          D3DBLEND_SRCALPHA,
2690          IDM_COLOR_SRC_BLEND_INV_SRC_ALPHA,      D3DBLEND_INVSRCALPHA,
2691          IDM_COLOR_SRC_BLEND_DEST_ALPHA,         D3DBLEND_DESTALPHA,
2692          IDM_COLOR_SRC_BLEND_INV_DEST_ALPHA,     D3DBLEND_INVDESTALPHA,
2693          IDM_COLOR_SRC_BLEND_DEST_COLOR,         D3DBLEND_DESTCOLOR,
2694          IDM_COLOR_SRC_BLEND_INV_DEST_COLOR,     D3DBLEND_INVDESTCOLOR,
2695          IDM_COLOR_SRC_BLEND_SRC_ALPHA_SAT,      D3DBLEND_SRCALPHASAT,
2696          IDM_COLOR_SRC_BLEND_BOTH_INV_SRC_ALPHA, D3DBLEND_BOTHINVSRCALPHA,
2697          IDM_COLOR_SRC_BLEND_FACTOR,             D3DBLEND_BLENDFACTOR,
2698          IDM_COLOR_SRC_BLEND_INV_FACTOR,         D3DBLEND_INVBLENDFACTOR,
2699          IDM_COLOR_DEST_BLEND_ZERO,              D3DBLEND_ZERO,
2700          IDM_COLOR_DEST_BLEND_ONE,               D3DBLEND_ONE,
2701          IDM_COLOR_DEST_BLEND_SRC_COLOR,         D3DBLEND_SRCCOLOR,
2702          IDM_COLOR_DEST_BLEND_INV_SRC_COLOR,     D3DBLEND_INVSRCCOLOR,
```

```
2703            IDM_COLOR_DEST_BLEND_SRC_ALPHA,         D3DBLEND_SRCALPHA,
2704            IDM_COLOR_DEST_BLEND_INV_SRC_ALPHA,     D3DBLEND_INVSRCALPHA,
2705            IDM_COLOR_DEST_BLEND_DEST_ALPHA,        D3DBLEND_DESTALPHA,
2706            IDM_COLOR_DEST_BLEND_INV_DEST_ALPHA,    D3DBLEND_INVDESTALPHA,
2707            IDM_COLOR_DEST_BLEND_DEST_COLOR,        D3DBLEND_DESTCOLOR,
2708            IDM_COLOR_DEST_BLEND_INV_DEST_COLOR,    D3DBLEND_INVDESTCOLOR,
2709            IDM_COLOR_DEST_BLEND_SRC_ALPHA_SAT,     D3DBLEND_SRCALPHASAT,
2710            IDM_COLOR_DEST_BLEND_FACTOR,            D3DBLEND_BLENDFACTOR,
2711            IDM_COLOR_DEST_BLEND_INV_FACTOR,        D3DBLEND_INVBLENDFACTOR,
2712            IDM_ALPHA_COMPOSITE_CLEAR,              E_COMPOSITE_CLEAR,
2713            IDM_ALPHA_COMPOSITE_SRC,                E_COMPOSITE_SRC,
2714            IDM_ALPHA_COMPOSITE_DEST,               E_COMPOSITE_DEST,
2715            IDM_ALPHA_COMPOSITE_SRC_OVER_DEST,      E_COMPOSITE_SRC_OVER_DEST,
2716            IDM_ALPHA_COMPOSITE_DEST_OVER_SRC,      E_COMPOSITE_DEST_OVER_SRC,
2717            IDM_ALPHA_COMPOSITE_SRC_IN_DEST,        E_COMPOSITE_SRC_IN_DEST,
2718            IDM_ALPHA_COMPOSITE_DEST_IN_SRC,        E_COMPOSITE_DEST_IN_SRC,
2719            IDM_ALPHA_COMPOSITE_SRC_ATOP_DEST,      E_COMPOSITE_SRC_ATOP_DEST,
2720            IDM_ALPHA_COMPOSITE_DEST_ATOP_SRC,      E_COMPOSITE_DEST_ATOP_SRC,
2721            IDM_ALPHA_COMPOSITE_SRC_OUT_DEST,       E_COMPOSITE_SRC_OUT_DEST,
2722            IDM_ALPHA_COMPOSITE_DEST_OUT_SRC,       E_COMPOSITE_DEST_OUT_SRC,
2723            IDM_ALPHA_COMPOSITE_SRC_XOR_DEST,       E_COMPOSITE_SRC_XOR_DEST,
2724            IDM_ALPHA_BLEND_OP_ADD,                 D3DBLENDOP_ADD,
2725            IDM_ALPHA_BLEND_OP_SUBTRACT,            D3DBLENDOP_SUBTRACT,
2726            IDM_ALPHA_BLEND_OP_REVERSE_SUBTRACT,    D3DBLENDOP_REVSUBTRACT,
2727            IDM_ALPHA_BLEND_OP_MINIMUM,             D3DBLENDOP_MIN,
2728            IDM_ALPHA_BLEND_OP_MAXIMUM,             D3DBLENDOP_MAX,
2729            IDM_ALPHA_SRC_BLEND_ZERO,               D3DBLEND_ZERO,
2730            IDM_ALPHA_SRC_BLEND_ONE,                D3DBLEND_ONE,
2731            IDM_ALPHA_SRC_BLEND_SRC_COLOR,          D3DBLEND_SRCCOLOR,
2732            IDM_ALPHA_SRC_BLEND_INV_SRC_COLOR,      D3DBLEND_INVSRCCOLOR,
2733            IDM_ALPHA_SRC_BLEND_SRC_ALPHA,          D3DBLEND_SRCALPHA,
2734            IDM_ALPHA_SRC_BLEND_INV_SRC_ALPHA,      D3DBLEND_INVSRCALPHA,
2735            IDM_ALPHA_SRC_BLEND_DEST_ALPHA,         D3DBLEND_DESTALPHA,
2736            IDM_ALPHA_SRC_BLEND_INV_DEST_ALPHA,     D3DBLEND_INVDESTALPHA,
2737            IDM_ALPHA_SRC_BLEND_DEST_COLOR,         D3DBLEND_DESTCOLOR,
2738            IDM_ALPHA_SRC_BLEND_INV_DEST_COLOR,     D3DBLEND_INVDESTCOLOR,
2739            IDM_ALPHA_SRC_BLEND_SRC_ALPHA_SAT,      D3DBLEND_SRCALPHASAT,
2740            IDM_ALPHA_SRC_BLEND_BOTH_INV_SRC_ALPHA, D3DBLEND_BOTHINVSRCALPHA,
2741            IDM_ALPHA_SRC_BLEND_FACTOR,             D3DBLEND_BLENDFACTOR,
2742            IDM_ALPHA_SRC_BLEND_INV_FACTOR,         D3DBLEND_INVBLENDFACTOR,
2743            IDM_ALPHA_DEST_BLEND_ZERO,              D3DBLEND_ZERO,
2744            IDM_ALPHA_DEST_BLEND_ONE,               D3DBLEND_ONE,
2745            IDM_ALPHA_DEST_BLEND_SRC_COLOR,         D3DBLEND_SRCCOLOR,
2746            IDM_ALPHA_DEST_BLEND_INV_SRC_COLOR,     D3DBLEND_INVSRCCOLOR,
2747            IDM_ALPHA_DEST_BLEND_SRC_ALPHA,         D3DBLEND_SRCALPHA,
2748            IDM_ALPHA_DEST_BLEND_INV_SRC_ALPHA,     D3DBLEND_INVSRCALPHA,
```

```
2749            IDM_ALPHA_DEST_BLEND_DEST_ALPHA,        D3DBLEND_DESTALPHA,
2750            IDM_ALPHA_DEST_BLEND_INV_DEST_ALPHA,    D3DBLEND_INVDESTALPHA,
2751            IDM_ALPHA_DEST_BLEND_DEST_COLOR,        D3DBLEND_DESTCOLOR,
2752            IDM_ALPHA_DEST_BLEND_INV_DEST_COLOR,    D3DBLEND_INVDESTCOLOR,
2753            IDM_ALPHA_DEST_BLEND_SRC_ALPHA_SAT,     D3DBLEND_SRCALPHASAT,
2754            IDM_ALPHA_DEST_BLEND_FACTOR,            D3DBLEND_BLENDFACTOR,
2755            IDM_ALPHA_DEST_BLEND_INV_FACTOR,        D3DBLEND_INVBLENDFACTOR
2756        };
2757        HMENU menu = TWS(::GetMenu(m_hWnd));
2758        for (UINT i = 0; i < NUM_OF(data); i++)
2759        {
2760            MENUITEMINFO info =
2761            {
2762                sizeof(info), MIIM_DATA
2763            };
2764            info.dwItemData = data[i].m_data;
2765            TWS(::SetMenuItemInfo(menu, data[i].m_id, FALSE, &info));
2766        }
2767    }
2768
2769    DWORD
2770    CMyD3DApplication::get_menu_data(DWORD id)
2771    {
2772        MENUITEMINFO info = { sizeof(info), MIIM_DATA };
2773        THR(::GetMenuItemInfo(TWS(::GetMenu(m_hWnd)), id, FALSE, &info));
2774        return DWORD(info.dwItemData);
2775    }
2776
2777    LRESULT
2778    CMyD3DApplication::on_left_button_down(HWND window, WPARAM, LPARAM lp, bool &han
2779    {
2780        ::SetCapture(window);
2781        m_scissor_rect.left = GET_X_LPARAM(lp);
2782        m_scissor_rect.top = GET_Y_LPARAM(lp);
2783        m_scissor_rect.right = GET_X_LPARAM(lp);
2784        m_scissor_rect.bottom = GET_Y_LPARAM(lp);
2785        THR(m_pd3dDevice->SetScissorRect(&m_scissor_rect));
2786        handled = true;
2787
2788        return LRESULT(0);
2789    }
2790
2791    LRESULT
2792    CMyD3DApplication::on_mouse_move(HWND window, WPARAM, LPARAM lp, bool &handled)
2793    {
2794        if (window == ::GetCapture())
```

```
2795        {
2796            m_scissor_rect.right = std::min(m_d3dsdBackBuffer.Width, UINT(GET_X_LPARAM(lp)));
2797            m_scissor_rect.bottom = std::min(m_d3dsdBackBuffer.Height, UINT(GET_Y_LPARAM(lp)))
2798            RECT tmp = m_scissor_rect;
2799            if (tmp.right < tmp.left)
2800            {
2801                std::swap(tmp.right, tmp.left);
2802            }
2803            if (tmp.bottom < tmp.top)
2804            {
2805                std::swap(tmp.bottom, tmp.top);
2806            }
2807            THR(m_pd3dDevice->SetScissorRect(&tmp));
2808            handled = true;
2809        }
2810        return LRESULT(0);
2811    }
2812
2813    LRESULT
2814    CMyD3DApplication::on_left_button_up(HWND window, WPARAM, LPARAM, bool &handled)
2815    {
2816        if (window == ::GetCapture())
2817        {
2818            TWS(::ReleaseCapture());
2819            handled = true;
2820        }
2821        return LRESULT(0);
2822    }
```