Chapter 4

2D Applications

> "There is no time like the present time."
>
> Tobias Smollett: *Humphrey Clinker*, 1771

4.1 Overview

This chapter describes how to use `IDirect3DDevice9` interface for a simple "two dimensional" application that only copies pixels. However, every Direct-3D application is going to use the methods and interfaces described in this application, not just "two dimensional" applications.

We start by examining the `IDirect3DSurface9` interface that Direct3D uses to expose collections of pixel data. We show how to create surfaces, fill them with data and use them in pixel copy operations in a simple demonstration application.

Next we discuss the `IDirect3DSwapChain9` interface that manages a collection of back buffers for presentation. Every device is created with a default swap chain, but new swap chains can also be created for multiple views in windowed mode.

Next, we discuss presentation. `Present` is one of the few `IDirect3DDevice9` methods where a failed `HRESULT` is part of normal practice. `Present` will fail when the device has been lost, leaving the application to regain the device at a later time.

Even though Direct3D applications can avoid GDI, they still need to respond to messages sent to the application's top-level window. We recommend strategies for a Direct3D application in responding to some of the messages. DirectX provides no direct way to combine GDI and Direct3D. However, GDI operations can be performed on a memory DC and the resulting pixel data used in a Direct3D application.

Finally, we discuss the video scan out portion of the pipeline and the presentation of images from the back buffers of swap chains onto the front buffer of the device. From there, the video scan out circuitry reads the data, applies a cursor overlay if a hardware cursor is used, gamma correction is applied, and the pixel data is converted to analog signals for the monitor.

4.2 Pixel Surfaces

Pixel surfaces are rectangular collections of pixel data. The memory layout of the pixel data is given by its D3DFORMAT. There are several places where surfaces are used on the device: back buffer surfaces, depth/stencil buffer surfaces, texture level surfaces, render target surfaces, and image surfaces.

Direct3D exposes a surface through the IDirect3DSurface9 interface, summarized in interface 4.1. Some device properties act as containers for surfaces and expose their contents by returning IDirect3DSurface9 interfaces to the application. An image surface can be created explicitly with the CreateOffscreenPlainSurface method. You can create surfaces in scratch memory, system memory, or device memory pools. The CreateDepthStencilSurface and CreateRenderTarget methods also return IDirect3DSurface9 interfaces for depth/stencil and render target surfaces discussed in chapter 5. A plain surface can't be the target of 3D rendering, but you can copy between plain surfaces and other surfaces.

```
HRESULT CreateOffscreenPlainSurface(UINT width,
            UINT height,
            D3DFORMAT format,
            D3DPOOL pool,
            IDirect3DSurface9 **result,
            HANDLE *unused);
```

CreateOffscreenPlainSurface will fail if the requested type of surface isn't supported on the device, or if there is insufficient memory in the system memory pool. Validate a surface format for use with CreateOffscreenPlainSurface by calling CheckDeviceFormat with the desired format and a resource type of D3DRTYPE_SURFACE. The unused argument must be NULL.

Interface 4.1: Summary of the IDirect3DSurface9 interface.

IDirect3DSurface9	
Read-Only Properties	
GetContainer	The containing resource or device.
GetDesc	A description of the contained pixel data.
GetDC	Creates a GDI device context for the surface.
Methods	
LockRect	Obtains direct access to the contained pixel data.
ReleaseDC	Releases the GDI device context for the surface.

UnlockRect	Releases direct access to the contained pixel data.

```
interface IDirect3DSurface9 : IDirect3DResource9
{
 //------------------------------------------------------------
 // read-only properties
 HRESULT GetContainer(REFIID container_iid,
            void **value);
 HRESULT GetDC(HDC **value);
 HRESULT GetDesc(D3DSURFACE_DESC *value);

 //------------------------------------------------------------
 // methods
 HRESULT LockRect(D3DLOCKED_RECT *data,
            const RECT *locked_region,
            DWORD flags);
 HRESULT ReleaseDC(HDC context);
 HRESULT UnlockRect();
};
```

For surfaces created with CreateOffscreenPlainSurface, GetContainer will only return success when container_iid is IID_Direct3DDevice9. Calls to GetContainer on surfaces returned by textures or cube textures succeed for the IIDs of their respective containers. GetDevice returns the associated device for all surfaces. IDirect3DSurface9 inherits from the IDirect3DResource9 interface, described in section 3.5.

The GetDesc method returns a description of the contained pixel data in a D3DSURFACE_DESC structure. The Format, Type, Usage, and Pool members are as described in section 2.7. MultiSampleType gives the multisampling used with a render target surface, as described in chapter 14. A surface created with CreateOffscreenPlainSurface will have Usage and MultiSampleType members set to zero, Type set to D3DRTYPE_SURFACE and Pool set to D3DPOOL_SYSTEMMEM.

```
typedef struct _D3DSURFACE_DESC
{
    D3DFORMAT           Format;
    D3DRESOURCETYPE     Type;
    DWORD               Usage;
    D3DPOOL             Pool;
    D3DMULTISAMPLE_TYPE MultiSampleType;
    DWORD MultiSampleQuality;
    UINT                Width;
    UINT                Height;
} D3DSURFACE_DESC;
```

4.3 Accessing Surface Pixel Data

To access the pixel data contained in a surface, use the LockRect and UnlockRect methods. A successful call to LockRect must be followed by a call to UnlockRect before the surface can be used with the device. A subrectangle of the surface can be locked, or it can be locked in its entirety when NULL is passed for the locked_region argument. The flags argument tells Direct3D how the data is to be used once the surface is locked and can be zero or more of the following flags:

```
#define D3DLOCK_DISCARD         0x00002000L
#define D3DLOCK_DONOTWAIT       0x00004000L
#define D3DLOCK_NO_DIRTY_UPDATE 0x00008000L
#define D3DLOCK_NOSYSLOCK       0x00000800L
#define D3DLOCK_READONLY        0x00000010L
```

D3DLOCK_DISCARD informs the runtime that the entire locked region will be written to but not read from. When a surface is locked with the discard flag, the runtime can proceed without providing a copy of the data for reading to the application. Without the discard flag, the runtime may be forced to flush any pending rendering operations on the pipeline before returning a copy of the surface data to the application. You can't use the discard flag in conjunction with a subregion; pass NULL for the subregion argument when using the discard flag.

The D3DLOCK_DONOTWAIT flag allows an application to determine if locking the surface would cause the runtime to block, waiting for pending rendering operations to complete. If the lock call would have blocked, then the method returns D3DERR_WASSTILLDRAWING and returns immediately without locking the surface. If the lock can be completed immediately, then the surface is locked normally.

Direct3D maintains a dirty region list for each managed surface that is used to minimize the amount of data that must be copied into the device when a resource is unlocked. A locked region doesn't affect the dirty region list if D3DLOCK_NO_DIRTY_UPDATE is used.

With D3DLOCK_READONLY, the application guarantees that no write operations will be performed on the data in the locked region. If an attempt is made to write into the locked region, the results are undefined.

D3DLOCK_NOSYSLOCK applies only to surfaces in video memory (default memory pool). In order to prevent a device from being lost while a video memory resource is locked, Direct3D obtains a system-wide critical section that prevents the device from being lost. It also blocks other parts of the operating system from executing, which can affect interactivity and responsiveness of the system. Specifying D3DLOCK_NOSYSLOCK prevents the system critical section from being taken. This flag is intended for lengthy lock operations such as a software renderer writing to a back buffer on a swap chain.

The LockRect method returns a D3DLOCKED_RECT structure defining the contained surface pixel data. Surface data is only guaranteed to be contiguous in memory along a scanline. The Pitch member defines the distance in bytes between adjacent scanlines. The pBits member points to the pixel data, beginning with the topmost scanline of the locked region. Writing beyond the end of the scanline, before the first scanline or after the last scanline of the region is undefined.

```
typedef struct _D3DLOCKED_RECT
{
    int  Pitch;
    void *pBits;
} D3DLOCKED_RECT;
```

When iterating over the pixels in a locked surface, it is very important to observe the Pitch and the size of the pixel data. The size of the pixel data is implied by its D3DFORMAT. A format of D3DFMT_A8 has a size of 8 bits, or one byte and can be represented by the standard Windows BYTE data type. A format of D3DFMT_A1R5G5B5 has a size of 16 bits, or two bytes and can be represented by the standard Windows WORD data type. A format of D3DFMT_A8R8G8B8 has a size of 32 bits, or four bytes and can be represented by the standard Windows DWORD data type. It is also the pixel format of D3DCOLOR, so that can also be used. D3DFMT_R8G8B8 has no convenient Windows data type of the same size, so you must use a BYTE for each color channel and perform pointer arithmetic in channels, not pixels.

The following code excerpt creates a 256x256 D3DFMT_A8R8G8B8 surface and fills it with a hue ramp. A scanline of D3DCOLOR is filled with a hue ramp, a loop over the scanlines in the surface replicates the constructed scanline over the surface with the Win32 ::CopyMemory routine.

```
// create an image surface
THR(m_pd3dDevice->CreateOffscreenPlainSurface(256, 256,
    D3DFMT_A8R8G8B8, D3DPOOL_SYSTEMMEM, &m_surface, NULL));

// create one scanline of the surface on the stack
D3DCOLOR scanline[256];
UINT i;
for (i = 0; i < 256; i++)
{
    float f = 0.5f + 0.5f*cosf(i*2.0f*D3DX_PI/255.0f);
    scanline[i] = hsv_d3dcolor(f, 1.0f, 1.0f); // h, s, v
}

// lock the surface to initialize it
D3DLOCKED_RECT lr;
THR(m_surface->LockRect(&lr, NULL, 0));
{
    BYTE *dest = static_cast<BYTE *>(lr.pBits);
    for (i = 0; i < 256; i++)
    {
        // copy scanline to the surface
        ::CopyMemory(dest, scanline, sizeof(scanline));
        dest += lr.Pitch;
    }
}
THR(m_surface->UnlockRect());
```

While IDirect3DSurface9 provides no methods for initializing surfaces from image files, or for converting surfaces between pixel formats, D3DX provides a variety of functions for these operations which are described in chapter 15.

If an application frequently locks surfaces and performs operations on the underlying pixel data, it may be convenient to define a helper class that locks a surface in its constructor and unlocks the surface in its destructor. This also ensures exception safety and guarantees that every successful LockRect is followed by an UnlockRect. The class can also provide accessors to avoid the error-prone scanline pointer arithmetic.

Listing 4.1 gives a surface lock helper class. Note that the helper switches the order of the locked_region and flags arguments when compared to the LockRect method and provides default values for these arguments. The helper assumes the more common case is to use flags other than zero when locking an entire surface instead of using flags of zero and locking a subrectangle of the surface.

Listing 4.1: <rt/surface.h>: A surface lock helper class.

```
#if !defined(RT_SURFACE_H)
#define RT_SURFACE_H
//-------------------------------------------------------------
// surface.h
//
// Helper functions for manipulating surfaces.
//
#include <atlbase.h>    // ATLASSERT(), CComPtr<>
#include <d3d9.h>       // IDirect3DSurface9

namespace rt {

//-------------------------------------------------------------
// surface_lock
//
// Helper class that locks a surface in its c'tor and unlocks
// it in its d'tor.  Provides accessors to the locked region.
//
class surface_lock
{
```

```
private:
    // Pitch is specified in bytes, not pixels.
    const BYTE *scanline(UINT y) const
    {
        return static_cast<const BYTE *>(m_data.pBits)
            + m_data.Pitch*y;
    }
    BYTE *scanline(UINT y)
    {
        return static_cast<BYTE *>(m_data.pBits) +
            m_data.Pitch*y;
    }

    CComPtr<IDirect3DSurface9> m_surface;
    D3DLOCKED_RECT m_data;

public:
    surface_lock(IDirect3DSurface9 *surface,
        DWORD flags = 0,
        const RECT *locked_region = NULL)
        : m_surface(surface)
    {
        THR(m_surface->LockRect(&m_data,
            locked_region, flags));
    }
    ˜surface_lock()
    {
        // destructors should never throw exceptions, so
        // we don’t use THR() here.  Also, we will never
        // be here unless the LockRect succeeded and
        // constructed a surface_lock, so UnlockRect
        // should always succeed.  We check anyway by
        // asserting success on the returned HRESULT.
        //
        // ATLASSERT is compiled out on optimized builds,
        // so use two statements because this:
        //      ATLASSERT(SUCCEEDED(m_surface->Unlock()))
        // would compile away the Unlock and introduce
        // a bug on a release build.
        const HRESULT hr = m_surface->UnlockRect(); hr;
        ATLASSERT(SUCCEEDED(hr));
    }

    // 8 bits per pixel: 1 pixel = 1 BYTE
    const BYTE *scanline8(UINT y) const
    {
        return scanline(y);
    }
    BYTE *scanline8(UINT y)
    {
        return scanline(y);
    }

    // 16 bits per pixel: 1 pixel = 1 WORD
    const WORD *scanline16(UINT y) const
    {
        return reinterpret_cast<const WORD *>(scanline(y));
    }
    WORD *scanline16(UINT y)
    {
        return reinterpret_cast<WORD *>(scanline(y));
    }

    // 24 bits per pixel: 1 pixel = 3 BYTEs
    const BYTE *scanline24(UINT y) const
    {
        return scanline(y);
    }
    BYTE *scanline24(UINT y)
    {
        return scanline(y);
    }

    // 32 bits per pixel: 1 pixel = 1 DWORD
    const DWORD *scanline32(UINT y) const
    {
        return reinterpret_cast<const DWORD *>(scanline(y));
    }
    DWORD *scanline32(UINT y)
    {
        return reinterpret_cast<DWORD *>(scanline(y));
    }
};  // surface_lock

};  // rt

#endif
```

4.4 Using GDI On A Surface

The GetDC and ReleaseDC methods on the surface interface allow you to use GDI on a surface whose Format is compatible with GDI. The only surface formats compatible with GDI are D3DFMT_R5G6B5, D3DFMT_X1R5G5B5, D3DFMT_R8G8B8, and D3DFMT_X8R8G8B8.

All the requirements for locking a surface apply to obtaining a GDI device context on the surface. Accordingly, GetDC will fail if:

1. The surface is already locked.
2. A device context for this surface has not been released.
3. The surface is contained in a texture and another surface in the texture is locked.
4. The surface is a render target that cannot be locked.
5. The surface is located in the default memory pool and was not created with the dynamic usage flag.
6. The surface is in the scratch pool.

The returned GDI device context is meant to be used for a few rendering operations on the surface through GDI and then immediately released. Once the device context has been created, a lock is held in the Direct3D runtime. This lock ensures that the runtime does not interfere with GDI rendering. Because of this lock, an application should release a GDI device context as soon as possible. In addition, the methods in the following table must not be called until the device context has been released. The restriction on Present applies only to swap chains containing the surface with the outstanding device context.

Interface	Method
IDirect3DCubeTexture9	LockRect
IDirect3DDevice9	ColorFill
	Present
	StretchRect
	UpdateSurface
	UpdateTexture
IDirect3DSurface9	LockRect
IDirect3DSwapChain9	Present
IDirect3DTexture9	LockRect

4.5 Swap Chains

Every device contains a set of default swap chains. The number of swap chains created with the device is returned by the GetNumberOfSwapChains method and the GetSwapChain method returns the swap chain interface for each of the swap chains in the default set. Only an adapter group device can be created with more than one swap chain. All devices can create additional swap chains after they have been created.

The characteristics of the default swap chain set are defined in the D3DPRESENT_PARAMETERS used to create the device. The swap chain consists of one, two or three back buffer surfaces and a front buffer surface. The front buffer surface is not directly accessible but still participates in the presentation of the swap chain. A back buffer surface is displayed on the monitor when Present is called, either on the device or on IDirect3DSwapChain9.

A device operating in exclusive mode uses its default swap chain for presentation. A device operating in windowed mode can use more than one swap chain, each presenting rendering results to its own window. An adapter group device in exclusive mode can present its rendering to multiple monitors in a coordinated manner through Present.

The CreateAdditionalSwapChain creates a new swap chain based on the given D3DPRESENT_PARAMETERS and returns an IDirect3DSwapChain9 interface. Note that a swap chain only contains back buffer surfaces and not a depth/stencil surface; the AutoDepthStencil and AutoDepthStencilFormat members of the presentation parameters are ignored by CreateAdditionalSwapChain. See chapter 5 for more on using depth/stencil buffers with a swap chain.

```
HRESULT CreateAdditionalSwapChain(D3DPRESENT_PARAMETERS *params,
            IDirect3DSwapChain9 **result);
```

The IDirect3DSwapChain9 interface is summarized in interface 4.2. The GetBackBuffer, GetDisplayMode, GetFrontBufferData and Present methods are similar for a swap chain and for a device, except that they apply only to a particular swap chain and not any swap chain on the device. The GetDevice method returns the device associated with this swap chain.

Interface 4.2: Summary of the IDirect3DSwapChain9 interface.

IDirect3DSwapChain9

Read-Only Properties	
GetBackBuffer	One of the back buffers of the swap chain.
GetDevice	Device associated with the swap chain.
GetDisplayMode	The video mode.
GetFrontBufferData	A copy of the front buffer.
GetPresentParameters	The presentation parameters.
GetRasterStatus	The raster scanout status.

Methods	
Present	Presents the next back buffer in the swap chain for display.

```
interface IDirect3DSwapChain9 : IUnknown
{
 //-----------------------------------------------------------
 // read-only properties
 HRESULT GetBackBuffer(UINT buffer,
            D3DBACKBUFFER_TYPE kind,
            IDirect3DSurface9 **value);
 HRESULT GetDevice(IDirect3DDevice9 **value);
 HRESULT GetDisplayMode(D3DDISPLAYMODE *value);
 HRESULT GetFrontBufferData(IDirect3DSurface9 *destination);
 HRESULT GetPresentParameters(D3DPRESENT_PARAMETERS *value);
 HRESULT GetRasterStatus(D3DRASTER_STATUS *value);

 //-----------------------------------------------------------
 // methods
 HRESULT Present(CONST RECT *source,
            CONST RECT *destination,
            HWND override,
            CONST RGNDATA *dirty_region,
            DWORD flags);
};
```

GetBackBuffer returns an interface pointer to one of the back buffer surfaces. The back buffers are numbered beginning with zero, with buffer zero being the buffer that will be displayed by the next call to Present, buffer one being displayed after buffer zero, and so-on. D3DBACKBUFFER_TYPE defines the type of back buffer to be retrieved. DirectX 9.0c does not support stereo rendering and the kind argument must always be D3DBACKBUFFER_TYPE_MONO.

```
typedef enum _D3DBACKBUFFER_TYPE
{
    D3DBACKBUFFER_TYPE_MONO  = 0,
    D3DBACKBUFFER_TYPE_LEFT  = 1,
    D3DBACKBUFFER_TYPE_RIGHT = 2
} D3DBACKBUFFER_TYPE;
```

The Present method performs the same function as the Present method on the device. It has an additional flags parameter that can be zero or more of the following values:

```
#define D3DPRESENT_DONOTWAIT      0x00000001L
#define D3DPRESENT_LINEAR_CONTENT 0x00000002L
```

The D3DPRESENT_DONOTWAIT flag instructs the method to return immediately with a failure result of D3DERR_WASSTILLDRAWING if presentation would cause the application to block before presentation could occur. The D3DPRESENT_LINEAR_CONTENT flag instructs the device that pixels in the source region should be converted from a linear color space to the sRGB color space during presentation. Support for linear to sRGB color space conversion on a device is indicated by the D3DCAPS3_LINEAR_TO_SRGB_PRESENTATION bit in the Caps3 member of D3DCAPS9.

4.6 Presentation

The contents of back buffers on a swap chain are made visible on the front buffer by calling Present. The front buffer is the source for pixel data read by the video scan out circuitry resulting in an image displayed on a monitor. If the D3DDEVCAPS_CANRENDERAFTERFLIP bit of D3DCAPS9::DevCaps is set, then the device can continue queuing rendering commands after a Present occurs, allowing for more parallelism between the device and CPU by allowing the next frame to be queued while the current frame is rendering. However, a device is not allowed to queue more than two frames of rendering.

```
#define D3DDEVCAPS_CANRENDERAFTERFLIP 0x00000800L
```

```
HRESULT Present(const RECT *source_rect,
            const RECT *dest_rect,
            HWND override_window,
            const RGNDATA *dirty_region);
```

The behavior of Present for a swap chain is defined by the SwapEffect member of the D3DPRESENT_PARAMETERS used to create the swap chain. SwapEffect can take on one of the values of the D3DSWAPEFFECT enumeration.

```
typedef enum _D3DSWAPEFFECT
{
    D3DSWAPEFFECT_DISCARD    = 1,
    D3DSWAPEFFECT_FLIP       = 2,
    D3DSWAPEFFECT_COPY       = 3
} D3DSWAPEFFECT;
```

In windowed mode, all swap effect semantics are implemented as copy operations. Swap chains created with an immediate presentation interval do not synchronize the copy operation with the monitor's vertical retrace and take effect immediately. A copy operation performed during the video scan out process can result in visible artifacts often described as "tearing" of the image. These artifacts can be avoided by synchronizing the copy operation with the video scan out process so that the copy does not take place if the video beam is located within the destination of the copy operation. Synchronizing presentation to the video refresh rate also ensures that frames will not be presented faster than the video refresh rate. If the video card does not support video beam location information, the copy happens immediately. See section 4.8.

	D3DSWAPEFFECT_DISCARD				D3DSWAPEFFECT_FLIP			
Action	Front	Back A	Back B	Back C	Front	Back A	Back B	Back C
Create	F	0 ?	1 ?	2 ?	F	0 ?	1 ?	2 ?
Draw	F	0 A	1 ?	2 ?	F	0 A	1 ?	2 ?
Present	A	2 ?	0 ?	1 ?	A	2 F	0 ?	1 ?
Draw	A	2 ?	0 B	1 ?	A	2 F	0 B	1 ?
Present	B	1 ?	2 ?	0 ?	B	1 F	2 A	0 ?
Draw	B	1 ?	2 ?	0 C	B	1 F	2 A	0 C
Present	C	0 ?	1 ?	2 ?	C	0 F	1 A	2 B
Draw	C	0 A	1 ?	2 ?	C	0 A	1 A	2 B
Present	A	2 ?	0 ?	1 ?	A	2 C	0 A	1 B

	D3DSWAPEFFECT_COPY		DDDSwapEffectCopyVSync	
Action	Front	Back A	Front	Back A
Create	F	0 ?	F	0 ?
Draw	F	0 A	F	0 A
Present	A	0 A	A	0 A
Draw	A	0 B	A	0 B
Present	B	0 B	B	0 B
Draw	B	0 C	B	0 C
Present	C	0 C	C	0 C
Draw	C	0 A	C	0 A
Present	A	0 A	A	0 A

Figure 4.1: The semantics of D3DSWAPEFFECT on a swap chain for an application drawing a repeating sequence of images A, B, C. Each entry contains a number denoting its back buffer index and a symbol denoting the buffer's contents after the action has taken place. "?" denotes an undefined surface, "F" denotes the initial contents of the front buffer.

The semantics of D3DSWAPEFFECT for a swap chain are summarized in figure 4.1. D3DSWAPEFFECT_DISCARD and D3DSWAPEFFECT_FLIP are are most easily depicted with the maximum number of back buffers; the results for fewer back buffers are similar. D3DSWAPEFFECT_COPY requires a single back buffer and always perform a copy operation. D3DSWAPEFFECT_DISCARD imposes the fewest semantics on Present: all back buffer contents are undefined after Present. This gives the device the most flexibility in meeting frame presentation semantics, providing for low overhead presentation. D3DSWAPEFFECT_FLIP is similar to the discard swap effect, but here the front buffer participates in the cycling of back buffers and the contents of the back buffers are preserved across Present. Meeting this requirement may cause the device to allocate additional buffers or perform additional copy operations during Present. Flip and discard swap effects are often used in exclusive mode.

Windowed	Exclusive
D3DPRESENT_INTERVAL_DEFAULT	D3DPRESENT_INTERVAL_DEFAULT
D3DPRESENT_INTERVAL_IMMEDIATE	D3DPRESENT_INTERVAL_IMMEDIATE
D3DPRESENT_INTERVAL_ONE	D3DPRESENT_INTERVAL_ONE
	D3DPRESENT_INTERVAL_TWO
	D3DPRESENT_INTERVAL_THREE
	D3DPRESENT_INTERVAL_FOUR

Table 4.1: Presentation intervals supported in windowed and exclusive mode.

In exclusive mode, the frequency of presentation is determined by the `FullScreen_PresentationInterval` member of the `D3DPRESENT_PARAMETERS` used to create the swap chain. The presentation interval specifies the maximum rate of presentation. Presentation can occur as fast as possible with `D3DPRESENT_INTERVAL_IMMEDIATE`, but this may involve tearing if the presentation occurs more rapidly than video scan out. The default presentation interval corresponds to the refresh rate of the adapter's video mode. Presentation can be synchronized to every 1, 2, 3, or 4 video refresh periods with the remaining enumerants. The presentation intervals supported by a particular device are given as a union of all supported presentation intervals in `D3DCAPS9::PresentationIntervals`.

```
#define D3DPRESENT_INTERVAL_DEFAULT   0x00000000L
#define D3DPRESENT_INTERVAL_ONE       0x00000001L
#define D3DPRESENT_INTERVAL_TWO       0x00000002L
#define D3DPRESENT_INTERVAL_THREE     0x00000004L
#define D3DPRESENT_INTERVAL_FOUR      0x00000008L
#define D3DPRESENT_INTERVAL_IMMEDIATE 0x80000000L
```

If the `dest_window` argument is not `NULL`, it specifies the window handle whose client region will be the target of the `Present`. If the `dest_window` argument is `NULL` and the `hDeviceWindow` member of the `D3DPRESENT_PARAMETERS` that created the swap chain is not `NULL`, then the `hDeviceWindow` member specifies the target of `Present`. If both `dest_window` and `hDeviceWindow` are `NULL`, then the swap chain is the default swap chain created with a device and the `focus_window` argument to `CreateDevice` is used as the target of `Present`.

The `source` and `dest` parameters can only be used with the copy swap effects and must be `NULL` for the flip and discard swap effects. A value of `NULL` for `source` or `dest` specifies the entire source or destination surface, respectively. With a copy swap effect, the source and destination rectangles are clipped against the source surface and destination window client area, respectively. A `::StretchBlt` operation is performed to copy the clipped source region to the clipped destination region.

The `dirty_region` parameter is only used with the copy swap effect and should be `NULL` for all other swap effects. With the copy swap effect, the dirty region allows the application to specify the minimal region of pixels within the source region that must be copied. The device will copy anywhere from this minimal region of the source rectangle up to the entire source rectangle and only uses the dirty region as an optimization hint.

4.7 Lost Devices and Reset

The returned HRESULT from Present is one of the few places where a failure code is expected as part of normal operation. Present will fail with D3DERR_DEVICELOST if the device has been lost. Once the device has been lost, all default pool resources must be freed before the device can be regained.

TestCooperativeLevel indicates the status of the device by returning D3DERR_DEVICELOST when the device is lost and cannot be regained, D3DERR_DEVICENOTRESET when the device was lost and can now be regained, or S_OK if the application has not lost the device. When the device can be regained, a call to Reset will restore the device, resources can be restored and the application can resume rendering.

```
HRESULT Reset(D3DPRESENT_PARAMETERS *params);
HRESULT TestCooperativeLevel();
```

Reset can also be used to change the values in the D3DPRESENT_PARAMETERS structure that defines the default swap chain. For instance, to support a toggle between windowed and exclusive mode, an application toggles the Windowed member of the presentation parameters, adjusts any other necessary data structures and calls Reset on the device.

4.8 Video Scan Out

The contents of the front buffer, resulting from Present, are read by the video scan out circuitry to create a video signal for the monitor. A description of the current display mode of the front buffer is returned by GetDisplayMode. The front buffer is not directly accessible, but a copy of the front buffer can be obtained with GetFrontBufferData. The destination argument must be an existing surface whose pixel dimensions are equal to the adapter's current display mode and whose format is D3DFMT_A8R8G8B8. The data is converted from the adapter's display mode format to the surface format during the copy.

```
HRESULT GetDisplayMode(D3DDISPLAYMODE *value);
HRESULT GetFrontBuffer(IDirect3DSurface9 *destination);
```

If the D3DCAPS_READ_SCANLINE bit of D3DCAPS9::Caps is set, then the device can report its video scan out scanline and vertical blank status.

```
#define D3DCAPS_READ_SCANLINE 0x00020000L
```

GetRasterStatus returns the video scan out status in a D3DRASTER_STATUS structure. The ScanLine member gives the current position of the raster beam, with zero being the topmost scanline in the frame. The InVBlank member is TRUE when the video beam is in vertical retrace from the bottom of the screen to the top.

```
HRESULT GetRasterStatus(D3DRASTER_STATUS *value);

typedef struct _D3DRASTER_STATUS
{
    BOOL InVBlank;
    UINT ScanLine;
} D3DRASTER_STATUS;
```

4.8.1 Cursor

In exclusive mode, Direct3D manages the cursor display. A hardware cursor can substitute the cursor image during video scan out. If a hardware cursor is not available, the runtime provides a software cursor through a read-modify-write operation on the front buffer. In windowed mode, an application can use either the GDI cursor or the Direct3D cursor. The Direct3D cursor can be shown or hidden with the ShowCursor method. ShowCursor does not return HRESULT, but instead returns the previous hide state of the cursor. If the return value is TRUE, then the cursor was visible before ShowCursor was called.

```
   BOOL ShowCursor(BOOL show);
   void SetCursorPosition(UINT x,
            UINT y,
            DWORD flags);
HRESULT SetCursorProperties(UINT hot_spot_x,
            UINT hot_spot_y,
            IDirect3DSurface9 *image);
```

The position of the cursor is set by calling SetCursorPosition. The flags argument can be zero or D3DCURSOR_IMMEDIATE_UPDATE to request that the cursor be refreshed at the rate of at least half the video refresh rate, but never faster than the video refresh rate. Without this flag, the cursor position may not change until the next call to Present. Using the flag prevents the visual state of the cursor from lagging too far behind user input when presentation rates are low. The x and y arguments specify the position of the cursor. In windowed mode, the position is in virtual desktop coordinates. In exclusive mode, the position is in screen space limited by the current display mode.

The cursor image can be moved relative to the position specified with SetCursorPosition by changing the cursor's hot spot. The hot spot is a coordinate relative to the top left of the cursor's image that corresponds to the point specified with SetCursorPosition. The hot spot and the cursor image can be set with SetCursorProperties. The image argument must be a D3DFMT_A8R8G8B8 surface whose pixel dimensions are smaller than the adapter's display mode. The dimensions must also be powers of two, although not necessarily identical. If the D3DCURSORCAPS_COLOR bit of D3DCAPS9::CursorCaps is set, the device supports a full color cursor in display modes with 400 or more scanlines. If the D3DCURSORCAPS_LOWRES bit is set, the device supports a full color cursor in display modes with less than 400 scanlines.

```
#define D3DCURSORCAPS_COLOR  0x00000001L
#define D3DCURSORCAPS_LOWRES 0x00000002L
```

4.8.2 Gamma Ramp

In exclusive mode, after the cursor has been applied, a gamma correcting CLUT can be applied to the pixel data before D/A conversion. In windowed mode, the application can use GDI for gamma correction as described in section 1.3. If the D3DCAPS2_FULLSCREENGAMMA bit of D3DCAPS9::Caps2 is set, the device supports a gamma ramp in exclusive mode.

```
#define D3DCAPS2_FULLSCREENGAMMA   0x00020000L
```

The gamma ramp property can be read with GetGammaRamp, returning a D3DGAMMARAMP structure.

```
void GetGammaRamp(D3DGAMMARAMP *value);
void SetGammaRamp(DWORD Flags,
        const D3DGAMMARAMP *value);

typedef struct _D3DGAMMARAMP
{
    WORD red[256];
    WORD green[256];
    WORD blue[256];
} D3DGAMMARAMP;
```

The gamma ramp property is set with SetGammaRamp and changes to the gamma ramp occur immediately without regard for the refresh rate. The flags argument indicates if the device should apply a calibration to the ramp with one of the following values.

```
#define D3DSGR_NO_CALIBRATION 0x00000000L
#define D3DSGR_CALIBRATE      0x00000001L
```

If the D3DCAPS2_CANCALIBRATEGAMMA bit of D3DCAPS9::Caps2 is set, then the device can apply a device specific calibration to the gamma ramp before setting it into the device.

```
#define D3DCAPS2_CANCALIBRATEGAMMA 0x00100000L
```

The following example shows how to compute the ramp values for a gamma-correcting ramp given the gamma of the monitor. As described in section 1.3, the gamma of a monitor can be measured interactively and this value used to create an appropriate gamma ramp for the device. The `rt_Gamma` sample demonstrates this technique for measuring the gamma and using it in the device's gamma ramp.

```
void
compute_ramp(D3DGAMMARAMP &ramp, float gamma)
{
    for (UINT i = 0; i < 256; i++)
    {
        const WORD val =
            static_cast<int>(65535*pow(i/255.f, 1.f/gamma));
        ramp.red[i] = val;
        ramp.green[i] = val;
        ramp.blue[i] = val;
    }
}
```

4.9 2D Pixel Copies

If we requested lockable back buffers as described in section 2.13, we could lock a rectangle of the back buffer and write into it directly with software. However, back buffer surfaces are device surfaces that reside in video memory. Accessing video memory directly with the CPU is an expensive operation and should be avoided. An image surface that resides in the system or scratch memory pools can be directly and quickly accessed by the CPU.

Direct3D considers three scenarios for copying rectangles of pixels: copying from device memory to device memory, copying from system memory to device memory and copying from device memory to system memory. The `StretchRect` method provides a way of efficiently copying pixels from one device memory surface to another. The `UpdateSurface` and `UpdateTexture` methods are tailored for moving data from system memory to device memory under application control and the `GetRenderTargetData` method is used to retrieve pixels from device memory into system memory.

Typically you would use `StretchRect` to compose a back buffer from images in an offscreen plain surface, or to move data between one device resource and another. `UpdateSurface` and `UpdateTexture` are useful when you need to update an image surface or texture resource in the default pool from data generated by the CPU. (Resources in the managed pool have their device resources updated automatically by the runtime when you modify the system memory shadow copy.) When you need to capture a screen shot or save rendered frames out for creating a movie file, you'll need to use `GetRenderTargetData`.

4.9.1 Pixel Copies Within Device Memory

StretchRect copies a rectangle of pixels from one device surface to another, possibly with stretching and filtering. StretchRect can copy an entire surface or subrectangles of a surface to a destination surface. The source and destination surface must be different surface objects. The two surfaces usually have the same D3DFORMAT, but StretchRect can also perform a limited form of color conversion during the copy. The source and destination surface can have different pixel dimensions.

```
HRESULT StretchRect(IDirect3DSurface9 *source,
            const RECT *source_rect,
            IDirect3DSurface9 *destination,
            const RECT *dest_rect,
            D3DTEXTUREFILTERTYPE filter);
```

When the source_rect parameter is NULL, the entire source surface is copied to the destination surface. When source_rect is not NULL, it points to a subrectangle of the source surface that is copied to the destination surface. Similarly, the dest_rect parameter gives the region into which the source pixels should be copied. A value of NULL causes the source pixels to be copied over the entire destination surface.

There are no size constraints between the source rectangle and the destination rectangle other than the pixel dimensions of the source and destination surfaces. StretchRect performs no clipping of source and destination rectangles and will fail if either the source rectangle or the corresponding destination rectangle lie outside the source or destination surfaces, respectively. StretchRect only performs a raw copy of pixel data; it does not perform any read-modify-write operations or interact with any device render states or texture stage states. Pixel copy operations involving transparency, rotation, filtering, stretching or other effects are best accomplished using the rendering pipeline with geometric primitives and textures.

The filter parameter specifies the filter to be used when resizing the source region to fit the destination region and can be D3DTEXF_NONE, D3DTEXF_POINT or D3DTEXF_LINEAR. The point and linear filters may be supported when minimizing or magnifying the source region. The following bit flags in the StretchRectFilterCaps member of the D3DCAPS9 structure describes the filtering support for StretchRect:

```
#define D3DPTFILTERCAPS_MAGFLINEAR 0x02000000L
#define D3DPTFILTERCAPS_MAGFPOINT  0x01000000L
#define D3DPTFILTERCAPS_MINFLINEAR 0x00000200L
#define D3DPTFILTERCAPS_MINFPOINT  0x00000100L
```

If the D3DDEVCAPS_CANBLTSYSTONONLOCAL bit of D3DCAPS9::DevCaps is set, then the device can perform StretchRect from system memory to non-local video memory, such as AGP memory.

TODO: How relevant is this CAPS bit anymore?

```
#define D3DDEVCAPS_CANBLTSYSTONONLOCAL 0x00020000L
```

Format Conversion With Device Pixel Copies

StretchRect can perform a color conversion operation when copying pixels. The supported conversions are from high-performance YUV surface formats to high-performance RGB surface formats. The exact format conversions supported are discovered by calling the CheckDeviceFormatConversion method on the IDirect3D9 interface. The method succeeds if the device supports a Present or StretchRect operation from the source format to the target format.

```
HRESULT CheckDeviceFormatConversion(UINT adapter,
            D3DDEVTYPE device_kind,
            D3DFORMAT source_fmt,
            D3DFORMAT target_fmt);
```

The adapter and device type parameters identify the device to be queried. The source format parameter must be either a FOURCC format or a valid back buffer format. The target format must be one of the following formats:

D3DFMT_A1R5G5B5
D3DFMT_X1R5G5B5
D3DFMT_R5G6B5
D3DFMT_R8G8B8
D3DFMT_A8B8G8R8
D3DFMT_A8R8G8B8
D3DFMT_X8B8G8R8
D3DFMT_X8R8G8B8
D3DFMT_A2R10G10B10
D3DFMT_A2B10G10R10
D3DFMT_A16B16G16R16
D3DFMT_A16B16G16R16F
D3DFMT_A32B32G32R32F

Device Pixel Copy Limitations

Because StretchRect operates on device memory directly, it is subject to a number of limitations and restrictions.

Stretch restrictions: 1. can't stretch when source and destination are the same surface 2. can't stretch from a render target surface to an offscreen plain surface 3. can't stretch on compressed formats 4. D3DDEVCAPS2_CAN_STRETCHRECT_FROM_TEXTURES if source is texture surface

Source/dest combinations:

DX8 Driver no stretching

Source	Destination			
	Texture	RT Texture	RT	Off-screen Plain
Texture	No	No	No	No
RT Texture	No	Yes	Yes	No
RT	No	Yes	Yes	No
Off-screen Plain	Yes	Yes	Yes	Yes

DX8 Driver stretching

Source	Destination			
	Texture	RT Texture	RT	Off-screen Plain
Texture	No	No	No	No
RT Texture	No	No	No	No
RT	No	Yes	Yes	No
Off-screen Plain	No	Yes	Yes	No

DX9 Driver no stretching

Source	Destination			
	Texture	RT Texture	RT	Off-screen Plain
Texture	No	Yes	Yes	No
RT Texture	No	Yes	Yes	No
RT	No	Yes	Yes	No
Off-screen Plain	No	Yes	Yes	Yes

DX9 Driver stretching

Source	Destination			
	Texture	RT Texture	RT	Off-screen Plain
Texture	No	Yes	Yes	No
RT Texture	No	Yes	Yes	No
RT	No	Yes	Yes	No
Off-screen Plain	No	Yes	Yes	No

Depth/stencil restrictions: 1. can't be textures 2. can't be discardable 3. entire surface must be copied 4. source and destination must be the same size 5. no filtering supported 6. cannot be called from within a scene

Downsampling multisample render target: 1. create multisample render target 2. create a non-multisampled render target of the same size 3. copy MS RT to non-MS RT

4.9.2 Copies From System Memory To Device Memory

You can use the CPU to directly fill any surface you can lock, but not all surfaces are lockable. Surfaces in device memory are not often lockable and access is slow when they are locked. Instead, the preferred approach is to update a system memory surface with the CPU and then use `UpdateSurface` or `UpdateTexture` to schedule a transfer of bits from system memory to device memory. The runtime queues the copy command along with the other rendering commands allowing the application to continue.

```
HRESULT UpdateSurface(IDirect3DSurface9 *source,
CONST RECT *source_rect,
IDirect3DSurface9 *destination,
CONST POINT *offset);
```

`UpdateSurface` transfers a rectangular region of pixels from the source surface to the destination surface. The `source_rect` parameter specifies the extent of the source surface that will be copied into the destination surface. If this parameter is `NULL`, then the entire source surface will be copied. The `offset` parameter gives the offset into the destination surface for the pixels that corresponds to the upper left corner of the source rectangle. If this parameter is `NULL`, then the upper left corner of the destination rectangle will be used. The function will fail if either the source rectangle or its shifted extent in the destination surface are outside the dimensions of the surfaces.

The source surface must be in the system memory pool and the destination surface must be in the default pool. The source and destination surfaces must

		Destination Formats			
		Texture	RT texture	RT	Plain
Source Formats	Texture	Yes	Yes	Yes	Yes
	RT texture	No	No	No	No
	RT	No	No	No	No
	Plain	Yes	Yes	Yes	Yes

Table 4.2: Combinations of source and destination surfaces supported with `UpdateSurface`.

have the same format, but they can be different sizes. `UpdateSurface` cannot be called while there is an outstanding GDI device context on the surface obtained from `GetDC`. `UpdateSurface` fails when either the source or destination surface is a surface created with multisampling or a depth stencil surface.

Surfaces that are contained within other resource types, render target surfaces and offscreen plain surfaces can be used with `UpdateSurface`. The supported combinations are given in table 4.2.

`UpdateTexture` is similar in function to `UpdateSurface`, but operates on an entire texture resource instead of a single surface. The dirty region maintained by the runtime for the source texture is used to determine the extent of the copy operation from system memory to device memory. See the discussion of each of the texture objects in chapter 11 for details on manipulating the dirty region of a texture.

```
HRESULT UpdateTexture(IDirect3DBaseTexture9 *source,
IDirect3DBaseTexture9 *destination);
```

When `UpdateTexture` is called, the accumulated dirty region since the last update is computed for level 0, the most detailed level of the texture. For mipmapped textures, the corresponding region of each mip level are considered dirty as well. The dirty region for a texture is an optimization hint and the driver may decide to copy more than just the dirty region.

`UpdateTexture` has similar restrictions to `UpdateSurface`. It will fail if the source texture is not in the system memory pool or if the destination texture is not in the default pool. The textures must be the same type (2D, cube, or volume) and format.

Level 0 of both texture must be the same size. The source texture cannot have fewer levels than the destination texture. If the source texture has more levels than the destination, then only the matching levels from the source are copied. If the destination texture has automatically generated mipmap levels, then level 0 of the source texture is copied to the destination and the destination mipmap levels are automatically regenerated. If the source texture has automatically generated mipmap levels, then the destination texture must also have automatically generated mipmap levels.

4.9.3 Copies From Device Memory To System Memory

There are only two ways to read back rendered images from the device: either create the device with a lockable back buffer or call GetRenderTargetData. Locking the back buffer is generally the slower of the two methods. GetRenderTargetData transfers the entire contents of the source render target surface to the destination surface.

```
HRESULT GetRenderTargetData(IDirect3DSurface9 *source,
IDirect3DSurface9 *destination);
```

The source and destination surfaces must be the same format and size. GetRenderTargetData fails if the source is multisampled or is not a render target surface or a level of a render target texture. GetRenderTargetData may return D3DERR_DRIVERINTERNALERROR or D3DERR_DEVICELOST with a proper set of parameters and its return value should be handled accordingly.

4.10 Filling Rectangles

If your application needs to fill a rectangle on a surface with a solid color, you can do this directly with the ColorFill method instead of locking and filling with the CPU. This is one way to easily initialize a surface to a solid color. To fill a surface with a pattern, you can render a textured quadrilateral and copy as needed.

```
HRESULT ColorFill(IDirect3DSurface9 *destination,
CONST RECT *region,
D3DCOLOR color);
```

If the region parameter is NULL, then the entire surface will be filled with the given color. The destination parameter must be a plain or render target surface in the default memory pool. The destination surface can be any format and the color value will be converted as needed. The only YUV surface formats supported by ColorFill on DirectX 7 and DirectX 8 level drivers are D3DFMT_UYVY and D3DFMT_YUY2.

4.11 Window Messages

The CreateDevice and Reset methods can generate windows messages during their execution. An application should not call device methods in response to messages generated during the execution of these methods. No methods should be called on the device until the device window has been fully constructed.

To reshape a device's default swap chain to new dimensions, the device must be Reset with new D3DPRESENT_PARAMETERS. To resize an additional swap chain, release the existing swap chain and create a new swap chain with the new D3DPRESENT_PARAMETERS. All references to default pool resources must be released before a device can be reset and need to be recreated after reset. Any other device state used will need to be explicitly restored to previous values. This could be an expensive operation to perform in response to dragging the window, but is reasonable once the final position has been selected. The `::StretchBlt` performed by presentation in windowed mode handles the disparity in size until the device is `Reset`. `Present`'s rectangle parameters can also be used to manage changes in aspect ratio and window size.

Applications such as real-time simulations and first-person games often use idle processing to continuous redraw the state of the simulation. The application's message loop is coded to avoid blocking when there are no messages waiting to be processed. Instead, the application continues to render new frames while awaiting for a message to arrive. Such applications need to respond properly to power management events or screen saver activation.

The following table gives a list of common windows messages and suggestions for handling them in a Direct3D application. This table is not a comprehensive list of all possible windows messages a Direct3D application will receive. Refer to the MSDN documentation for a comprehensive listing of applicable messages. The SDK sample framework follows most of these suggestions, see appendix A.

`WM_ACTIVATEAPP` Sent when the active window changes between applications. Suspend or resume continuous redraw.

`WM_CLOSE` Sent to signal application termination. Release all objects on the device, release the device and exit. When closing a window used with a swap chain, release the swap chain.

`WM_COMPACTING` Sent to indicate a low memory condition in the system. Release all resources not currently in use.

`WM_CONTEXTMENU` Sent when the user clicks the context button in the window. In windowed mode, handle popup menus.

`WM_CREATE` Sent to a window while it is being created. The `WM_CREATE` message is sent to a window before the corresponding `::CreateWindow` call has completed. You should not construct a device in response to `WM_CREATE`, but at some point after the corresponding call to `::CreateWindow` returns.

`WM_DISPLAYCHANGE` Sent when the display resolution of the desktop has changed. The device may have been lost as a result of the change. Reshape the swap chain.

`WM_ENTERMENULOOP` Sent when a modal menu loop is entered. Pause continuous redraw when using menus.

`WM_ENTERSIZEMOVE` Sent when starting a window size or move operation. Suspend generation of new frames while the user begins a resize or move operation on the window's frame.

`WM_ERASEBKGND` Sent when the window's background needs erasing. Return `TRUE` to indicate that the background has been erased.

WM_EXITMENULOOP Sent when the modal menu loop is exited. Resume continuous redraw when the menu is no longer in use.

WM_EXITSIZEMOVE Sent after a window size or move operation has completed. Reshape the swap chain.

WM_GETMINMAXINFO Sent when the size or position of a window is about to change. An application can enforce aspect ratio or other size constraints.

WM_MOUSEMOVE Sent when the mouse moves. If using Direct3D's cursor, make the cursor follow the mouse with SetCursorPosition.

WM_NCHITTEST Sent when the mous moves or a mouse button is pressed or released. Prevent menu selection in exclusive mode.

WM_PAINT Sent to repaint damaged portions of a window. Respond to paint messages by rendering the scene, if necessary, and presenting the back buffer.

WM_POWERBROADCAST Sent when a power management event is generated. Suspend or resume the application. An application should always allow the system to enter sleep mode to conserve power by properly implementing suspend and resume logic in its message loop.

WM_SETCURSOR Sent to set the cursor on a window. Turn off Win32 cursor and use Direct3D cursor in exclusive mode.

WM_SHOWWINDOW Sent when the window is about to be hidden or shown. Suspend or resume continuous redraw.

WM_SIZE Sent after the size of a window has changed. Check for minimization or hiding of the application's window. Reshape the swap chain.

WM_SIZING Sent while a window is being resized. If the application is dynamically refreshing during a resize operation, render into the back buffer and present normally. Enforce aspect ratio or other size constraints by modifying the allowed window size. Reshape the swap chain.

WM_SYSCOMMAND Sent during a system command. When the screen saver is activated or the display is powering down, this indicates an idle situation and the application should suspend continuous redraw. Disable moving or resizing the window in exclusive mode.

4.12 rt_2DApp Sample Application

The sample application listed here creates a hue ramp in a D3DFMT_A8R8G8B8 image surface and uses StretchRect to draw each frame. A list of subrectangles is constructed to replicate a single tile surface across the entire back buffer with one call to StretchRect.

The DirectX AppWizard was used to create the sample. Only the sample-specific source file rt_2DApp.cpp is listed here. See appendix A for a description of the DirectX AppWizard and the SDK sample framework.

Listing 4.2: rt_2DApp.cpp: A simple 2D application using StretchRect.

```
////////////////////////////////////////////////////////////
// rt_2DApp.cpp
//
// A simple demonstration of 2D application capabilities in
// Direct3D

// C++ includes
#include <algorithm>
#include <sstream>
#include <vector>

// Win32 includes
#define STRICT
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <basetsd.h>
#include <commdlg.h>
#include <commctrl.h>

// ATL includes
#include <atlbase.h>

// Direct3D includes
#include <d3dx9.h>
#include <dxerr9.h>

// SDK framework includes
#include "DXUtil.h"
#include "D3DEnumeration.h"
#include "D3DSettings.h"
#include "D3DApp.h"
#include "D3DFont.h"
#include "D3DUtil.h"

// rt includes
#include "rt/app.h"
#include "rt/hr.h"
#include "rt/hsv.h"
#include "rt/mat.h"
```

```
#include "rt/media.h"
#include "rt/misc.h"
#include "rt/rtgdi.h"
// rt smart surface lock; comment this out for manual locking
#include "rt/surface.h"
#include "rt/tstring.h"

// sample includes
#include "resource.h"
#include "rt_2DApp.h"

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

//////////////////////////////////////////////////////////////
// WinMain()
//
// Entry point to the program. Initializes everything, and
// goes into a message-processing loop. Idle time is used to
// render the scene.
//
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, INT)
{
    CMyD3DApplication d3dApp;

    g_pApp  = &d3dApp;
    g_hInst = hInst;

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

    return d3dApp.Run();
}

//////////////////////////////////////////////////////////////
// CMyD3DApplication()
//
// Application constructor.  Paired with ~CMyD3DApplication()
// Member variables should be initialized to a known state
// here.  The application window has not yet been created
// and no Direct3D device has been created, so any
// initialization that depends on a window or Direct3D should
// be deferred to a later stage.
//
CMyD3DApplication::CMyD3DApplication() :
    CD3DApplication(),
    m_device_tile(),
    m_tile_width(256),
    m_tile_height(256),
    m_system_tile(),
    m_stretch(false),
    m_capture_front(false),
    m_capture_back(false),
    m_magnify(false),
    m_filter(D3DTEXF_NONE),
    m_capture_file(_T("")),
    m_background_file(_T("")),
    m_background(BACKGROUND_HUE_RAMP),
    m_fill_colors(false),
    m_statistics(true),
    m_dialogs(false),
    m_draw_sprites(true),
    m_sprite(),
    m_sprite_state(),
    m_sprite_file(rt::find_media(_T("banana.bmp"))),
    m_sprite_texture(),
    m_sprite_xform(1, 0, 0, 0,
                   0, 1, 0, 0,
                   0, 0, 1, 0,
                   0, 0, 0, 1),
    m_bLoadingApp(TRUE),
    m_font(_T("Arial"), 12, D3DFONT_BOLD)
{
    m_dwCreationWidth           = 500;
    m_dwCreationHeight          = 375;
    m_strWindowTitle            = TEXT("rt_2DApp");
    m_d3dEnumeration.AppUsesDepthBuffer   = TRUE;
        m_bStartFullscreen                      = false;
        m_bShowCursorWhenFullscreen     = false;

    // Read settings from registry
    ReadSettings();
}

////////////////////////////////////////////////////////////
// ~CMyD3DApplication()
//
// Application destructor.  Paired with CMyD3DApplication()
```

```
//
CMyD3DApplication::~CMyD3DApplication()
{
}

////////////////////////////////////////////////////////////
// OneTimeSceneInit()
//
// Paired with FinalCleanup().  The window has been created
// and the IDirect3D9 interface has been created, but the
// device has not been created yet.  Here you can perform
// application-related initialization and cleanup that does
// not depend on a device.
//
HRESULT CMyD3DApplication::OneTimeSceneInit()
{
    // Drawing loading status message
    ::SendMessage(m_hWnd, WM_PAINT, 0, 0);
    m_bLoadingApp = FALSE;
    return S_OK;
}

////////////////////////////////////////////////////////////
// FinalCleanup()
//
// Paired with OneTimeSceneInit().  Called before the app
// exits, this function gives the app the chance to cleanup
// after itself.
//
HRESULT CMyD3DApplication::FinalCleanup()
{
    // Write the settings to the registry
    WriteSettings();
    return S_OK;
}

////////////////////////////////////////////////////////////
// ReadSettings()
//
// Read the app settings from the registry
//
void CMyD3DApplication::ReadSettings()
{
    HKEY hkey;
    if (ERROR_SUCCESS == ::RegCreateKeyEx(HKEY_CURRENT_USER,
        DXAPP_KEY, 0, NULL, REG_OPTION_NON_VOLATILE, KEY_READ,
        NULL, &hkey, NULL))
    {
        // Read the stored window width/height.
        ::DXUtil_ReadIntRegKey(hkey, TEXT("Width"),
            &m_dwCreationWidth, m_dwCreationWidth);
        ::DXUtil_ReadIntRegKey(hkey, TEXT("Height"),
            &m_dwCreationHeight, m_dwCreationHeight);
        ::RegCloseKey(hkey);
    }
}

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

    if (ERROR_SUCCESS == ::RegCreateKeyEx(HKEY_CURRENT_USER,
        DXAPP_KEY, 0, NULL, REG_OPTION_NON_VOLATILE, KEY_WRITE,
        NULL, &hkey, NULL))
    {
        // Write the window width/height.
        ::DXUtil_WriteIntRegKey(hkey, TEXT("Width"),
            m_rcWindowClient.right);
        ::DXUtil_WriteIntRegKey(hkey, TEXT("Height"),
            m_rcWindowClient.bottom);
        ::RegCloseKey(hkey);
    }
}

/////////////////////////////////////////////////////////////
// InitDeviceObjects()
//
// Paired with DeleteDeviceObjects().  The device has been
// created.  Resources that are not lost on Reset() can be
// created here -- resources in D3DPOOL_MANAGED,
// D3DPOOL_SCRATCH, or D3DPOOL_SYSTEMMEM.  Vertex shaders
// and pixel shaders can also be created here as they are
// not lost on Reset().
//
HRESULT CMyD3DApplication::InitDeviceObjects()
{
    init_background();
    THR(::D3DXCreateSprite(m_pd3dDevice, &m_sprite));
    init_sprite();
    m_font.InitDeviceObjects(m_pd3dDevice);
    return S_OK;
}

////////////////////////////////////////////////////////////
// DeleteDeviceObjects()
//
// Paired with InitDeviceObjects().  Called when the app
// is exiting, or the device is being changed, this function
// deletes any device dependent objects.
//
HRESULT CMyD3DApplication::DeleteDeviceObjects()
{
    m_system_tile = 0;
    m_sprite = 0;
    m_sprite_texture = 0;
    m_font.DeleteDeviceObjects();
    return S_OK;
}

////////////////////////////////////////////////////////////
// RestoreDeviceObjects()
//
// Paired with InvalidateDeviceObjects().  The device exists,
// but may have just been Reset().  Resources in D3DPOOL_DEFAULT
// and any other device state that persists during rendering
// should be set here.  Render states, matrices, textures, etc.,
// that don't change during rendering can be set once here to
// avoid redundant state setting during Render() or FrameMove().
//
HRESULT CMyD3DApplication::RestoreDeviceObjects()
{
    // is the background tile magnified?
    m_magnify = (m_tile_width < m_d3dsdBackBuffer.Width) ||
        (m_tile_height < m_d3dsdBackBuffer.Height);
    // set stretch rect filter menu item state
    HMENU menu = ::GetMenu(m_hWnd);
    rt::check_menu(menu, ID_STRETCHFILTER_NONE, false);
    rt::check_menu(menu, ID_STRETCHFILTER_POINT, false);
    rt::check_menu(menu, ID_STRETCHFILTER_LINEAR, false);
    rt::enable_menu(menu, ID_STRETCHFILTER_POINT, true);
    rt::enable_menu(menu, ID_STRETCHFILTER_LINEAR, true);
    if (m_magnify)
    {
        if (!(m_d3dCaps.StretchRectFilterCaps & D3DPTFILTERCAPS_MAGFPOINT))
        {
            if (D3DTEXF_POINT == m_filter)
            {
                m_filter = D3DTEXF_NONE;
            }
            rt::enable_menu(menu, ID_STRETCHFILTER_POINT, false);
        }
        if (!(m_d3dCaps.StretchRectFilterCaps & D3DPTFILTERCAPS_MAGFLINEAR))
        {
            if (D3DTEXF_LINEAR == m_filter)
            {
                m_filter = D3DTEXF_NONE;
            }
            rt::enable_menu(menu, ID_STRETCHFILTER_LINEAR, false);
        }
    }
    else
    {
        if (!(m_d3dCaps.StretchRectFilterCaps & D3DPTFILTERCAPS_MINFPOINT))
        {
            if (D3DTEXF_POINT == m_filter)
            {
                m_filter = D3DTEXF_NONE;
            }
            rt::enable_menu(menu, ID_STRETCHFILTER_POINT, false);
        }
        if (!(m_d3dCaps.StretchRectFilterCaps & D3DPTFILTERCAPS_MINFLINEAR))
        {
            if (D3DTEXF_LINEAR == m_filter)
            {
                m_filter = D3DTEXF_NONE;
            }
            rt::enable_menu(menu, ID_STRETCHFILTER_LINEAR, false);
        }
    }
    rt::check_menu(menu, ID_STRETCHFILTER_NONE + m_filter, true);

    // can we display GDI dialogs?
    m_dialogs = ((D3DFMT_X8R8G8B8 == m_d3dpp.BackBufferFormat) ||
                 (D3DFMT_R5G6B5 == m_d3dpp.BackBufferFormat) ||
                 (D3DFMT_X1R5G5B5 == m_d3dpp.BackBufferFormat)) &&
        (D3DMULTISAMPLE_NONE == m_d3dsdBackBuffer.MultiSampleType) &&
        (D3DPRESENTFLAG_LOCKABLE_BACKBUFFER & m_d3dpp.Flags) &&
        (D3DSWAPEFFECT_DISCARD == m_d3dpp.SwapEffect) &&
        !(D3DCREATE_ADAPTERGROUP_DEVICE & m_dwCreateFlags);
    if (m_dialogs)
    {
        THR(m_pd3dDevice->SetDialogBoxMode(true));
    }

    m_font.RestoreDeviceObjects();
    restore_background();
    restore_sprite();

    rt::enable_menu(menu, ID_OPTIONS_DRAWSPRITES, m_sprite_texture != 0);

    return S_OK;
}

void
CMyD3DApplication::restore_sprite()
{
    THR(m_sprite->OnResetDevice());
    const UINT NUM_SPRITES = 10;
    const float SPRITE_SIZE = 64.f;
    m_sprite_state.resize(NUM_SPRITES);
    const float scale = 2.f*D3DX_PI/(NUM_SPRITES-1);
    for (UINT s = 0; s < NUM_SPRITES; s++)
    {
        const float cx = m_d3dsdBackBuffer.Width/2.f;
        const float cy = m_d3dsdBackBuffer.Height/2.f;
        const float x = cx*(1.f + 0.5f*cosf(s*scale)) - SPRITE_SIZE*0.5f;
        const float y = cy*(1.f + 0.5f*sinf(s*scale)) - SPRITE_SIZE*0.5f;
        m_sprite_state[s].m_position = D3DXVECTOR3(x, y, 0.0f);
        m_sprite_state[s].m_color =
            D3DCOLOR_ARGB(32 + (255-32)*s/(NUM_SPRITES-1), 255, 255, 255);
    }
}

////////////////////////////////////////////////////////////
// InvalidateDeviceObjects()
//
// Invalidates device objects.  Paired with
// RestoreDeviceObjects()
//
HRESULT CMyD3DApplication::InvalidateDeviceObjects()
{
    if (BACKGROUND_GDI_ELLIPSE == m_background)
    {
        m_system_tile = 0;
    }
    m_device_tile = 0;
    m_font.InvalidateDeviceObjects();

    // might not be able to call this after Reset
    if (m_dialogs)
    {
        THR(m_pd3dDevice->SetDialogBoxMode(false));
    }

    m_sprite_state.clear();
    THR(m_sprite->OnLostDevice());

    return S_OK;
}

/////////////////////////////////////////////////////////////
// Render()
//
// Called once per frame, the call is the entry point for 3d
// rendering. This function sets up render states, clears the
// viewport, and renders the scene.
//
HRESULT CMyD3DApplication::Render()
{
    // copy the tile all over the back buffer
    const UINT width = m_d3dsdBackBuffer.Width;
    const UINT height = m_d3dsdBackBuffer.Height;
    CComPtr<IDirect3DSurface9> back;
    THR(m_pd3dDevice->GetBackBuffer(0, 0,
        D3DBACKBUFFER_TYPE_MONO, &back));
    if (m_stretch)
    {
        THR(m_pd3dDevice->StretchRect(m_device_tile, NULL,
            back, NULL, m_filter));
    }
    else
    {
        for (UINT y = 0; y < height; y += m_tile_height)
        {
            for (UINT x = 0; x < width; x += m_tile_width)
            {
                const RECT src =
                {
                    0, 0,
                    x + m_tile_width <= width ?
                        m_tile_width : width-x,
                    y + m_tile_height <= height ?
                        m_tile_height : height-y
                };
                const RECT dest =
                {
                    x, y,
                    x + src.right, y + src.bottom
                };
                THR(m_pd3dDevice->StretchRect(m_device_tile,
                    &src, back, &dest, m_filter));
            }
        }
    }

    // draw rainbow circle of squares
    if (m_fill_colors)
    {
        const UINT cx = m_d3dsdBackBuffer.Width/2;
        const UINT cy = m_d3dsdBackBuffer.Height/2;
        const UINT radius = (cx < cy ? cx : cy) - 8;
        const UINT num_fills = 64;
        const float scale = 2.f*D3DX_PI/float(num_fills-1);
        for (UINT i = 0; i < num_fills; i++)
        {
            const UINT x = cx + UINT(radius*cosf(i*scale));
            const UINT y = cy + UINT(radius*sinf(i*scale));
            const RECT dest = { x-4, y-4, x+4, y+4 };
            const float hue = 0.5f + 0.5f*cosf(i*3.f*scale);
            THR(m_pd3dDevice->ColorFill(back, &dest,
                rt::hsv(hue, 0.5f, 1.f)));
        }
    }

    THR(m_pd3dDevice->BeginScene());
    if (m_draw_sprites)
    {
        THR(m_sprite->Begin(D3DXSPRITE_ALPHABLEND));
                THR(m_sprite->SetTransform(rt::anon(rt::mat_scale(0.75f))));
        for (size_t s = 0; s < m_sprite_state.size(); s++)
        {
                        D3DXVECTOR3 pos = m_sprite_state[s].m_position;
                        THR(m_sprite->Draw(m_sprite_texture, NULL, NULL,
                                &m_sprite_state[s].m_position, m_sprite_state[s].m_color))
        }
        THR(m_sprite->End());
    }
    // Render stats and help text
    if (m_statistics)
    {
        RenderText();
    }
    THR(m_pd3dDevice->EndScene());

    // capture front buffer to a file
    if (m_capture_front)
    {
        CComPtr<IDirect3DSurface9> front;
        THR(m_pd3dDevice->CreateOffscreenPlainSurface(
            m_d3dSettings.Windowed_DisplayMode.Width,
            m_d3dSettings.Windowed_DisplayMode.Height,
            D3DFMT_A8R8G8B8, D3DPOOL_SYSTEMMEM, &front,
            NULL));
        THR(m_pd3dDevice->GetFrontBufferData(0, front));
        THR(::D3DXSaveSurfaceToFile(m_capture_file.c_str(),
            D3DXIFF_BMP, front, NULL, NULL));
        m_capture_front = false;
    }
    // capture back buffer to a file
    else if (m_capture_back)
    {
        THR(::D3DXSaveSurfaceToFile(m_capture_file.c_str(),
            D3DXIFF_BMP, back, NULL, NULL));
        m_capture_back = false;
    }

    return S_OK;
}

////////////////////////////////////////////////////////////
// RenderText()
//
// Renders stats and help text to the scene.
//
HRESULT CMyD3DApplication::RenderText()
{
    const D3DCOLOR yellow = D3DCOLOR_ARGB(255,255,255,0);
    m_font.DrawText(2, 20.0f, yellow, m_strDeviceStats);
    m_font.DrawText(2, 0.0f, yellow, m_strFrameStats);
    m_font.DrawText(2, m_d3dsdBackBuffer.Height - 20.0f,
        yellow, TEXT("Press 'F2' to configure display"));
    return S_OK;
}
```

```

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

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

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

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

//////////////////////////////////////////////////////////////
// on_command
//
// WM_COMMAND message handler
//
LRESULT
CMyD3DApplication::on_command(HWND window, WPARAM wp,
                              LPARAM, bool &handled)
{
    const UINT control = LOWORD(wp);
    HMENU menu = ::GetMenu(window);
    handled = true;
    switch (control)
    {
    case ID_OPTIONS_DRAWSPRITES:
        rt::toggle_menu(menu, control, m_draw_sprites);
        break;

    case ID_OPTIONS_SPRITEIMAGE:
        if (get_sprite_filename())
        {
            m_sprite_texture = 0;
            init_sprite();
            rt::enable_menu(menu, ID_OPTIONS_DRAWSPRITES, m_sprite_texture != 0)
        }
        break;

    case ID_BACKGROUND_HUERAMP:
        rt::check_menu(menu, ID_BACKGROUND_HUERAMP + m_background, false);
        m_background = BACKGROUND_HUE_RAMP;
        rt::check_menu(menu, ID_BACKGROUND_HUERAMP + m_background, true);
        recreate_background();
        break;

    case ID_BACKGROUND_IMAGE:
        if (get_background_filename())
        {
            rt::check_menu(menu, ID_BACKGROUND_HUERAMP + m_background, false);
            m_background = BACKGROUND_IMAGE;
            rt::check_menu(menu, ID_BACKGROUND_HUERAMP + m_background, true);
            recreate_background();
        }
        break;

    case ID_BACKGROUND_GDIELLIPSE:
        rt::check_menu(menu, ID_BACKGROUND_HUERAMP + m_background, false);
        m_background = BACKGROUND_GDI_ELLIPSE;
        rt::check_menu(menu, ID_BACKGROUND_HUERAMP + m_background, true);
        recreate_background();
        break;

    case ID_FILE_SAVEBACK:
        m_capture_back = get_save_filename();
```

```
            break;

        case ID_FILE_SAVEFRONT:
            m_capture_front = get_save_filename();
            break;

        case ID_OPTIONS_STATISTICS:
            rt::toggle_menu(menu, control, m_statistics);
            break;

        case ID_OPTIONS_STRETCHBACKGROUND:
            rt::toggle_menu(menu, control, m_stretch);
            break;

        case ID_OPTIONS_FILLCOLORS:
            rt::toggle_menu(menu, control, m_fill_colors);
            break;

        case ID_STRETCHFILTER_NONE:
        case ID_STRETCHFILTER_POINT:
        case ID_STRETCHFILTER_LINEAR:
            rt::check_menu(menu,
                ID_STRETCHFILTER_NONE + m_filter, false);
            m_filter = D3DTEXTUREFILTERTYPE(control -
                ID_STRETCHFILTER_NONE);
            rt::check_menu(menu,
                ID_STRETCHFILTER_NONE + m_filter, true);
            break;

        default:
            handled = false;
        }

        return 0;
}

////////////////////////////////////////////////////////////
// get_save_filename
//
// Gets the filename for saving the front or back buffer.
//
bool
CMyD3DApplication::get_save_filename()
{
    rt::pauser pause(*this);
    TCHAR buffer[MAX_PATH] = { 0 };
        OPENFILENAME ofn =
        {
            sizeof(ofn), NULL, NULL,
            _T("Bitmap files (*.bmp)\0")
                _T("*.bmp\0")
            _T("All files (*.*)\0")
                _T("*.*\0")
            _T("\0"), NULL, 0, 1, buffer, NUM_OF(buffer),
            NULL, 0, NULL, NULL,
            OFN_PATHMUSTEXIST | OFN_CREATEPROMPT
        };
        if (!m_dialogs && !m_bWindowed)
        {
            THR(ToggleFullscreen());
        }
        if (::GetSaveFileName(&ofn))
        {
            m_capture_file = buffer;
            return true;
        }
        return false;
}

////////////////////////////////////////////////////////////
// get_background_filename
//
// Open an image file for reading as the background image.
//
bool
CMyD3DApplication::get_background_filename()
{
        rt::pauser pause(*this);
        TCHAR buffer[MAX_PATH] = { 0 };
        OPENFILENAME ofn =
        {
            sizeof(ofn), m_hWnd, NULL,
            _T("All image files\0")
                _T("*.bmp;*.dib;*.jpg;*.jpeg;*.png;")
                _T("*.dds;*.tga;*.pbm;*.pgm;*.ppm;*.pnm\0")
            _T("Bitmap images (*.bmp,*.dib)\0")
                _T("*.bmp;*.dib\0")
            _T("JPEG images (*.jpg,*.jpeg)\0")
                _T("*.jpg;*.jpeg\0")
            _T("PNG images (*.png)\0")
                _T("*.png\0")
            _T("DDS images (*.dds)\0")
```

```
            _T("*.dds\0")
        _T("Targa images (*.tga)\0")
            _T("*.tga\0")
        _T("PNM images (*.p[bgpn]m)\0")
            _T("*.pbm;*.pgm;*.ppm;*.pnm\0")
        _T("All files (*.*)\0")
            _T("*.*\0")
        _T("\0"), NULL, 0, 1, buffer, NUM_OF(buffer),
        NULL, 0, NULL, NULL,
        OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST
    };
    if (!m_dialogs && !m_bWindowed)
    {
        THR(ForceWindowed());
    }
    if (::GetOpenFileName(&ofn))
    {
        m_background_file = buffer;
        return true;
    }
    return false;
}

////////////////////////////////////////////////////////////
// hue_ramp
//
// Construct a hue ramp on a scanline.
//
void
hue_ramp(D3DCOLOR *scanline, UINT width)
{
    for (UINT i = 0; i < width; i++)
    {
        scanline[i] = rt::hsv(i/float(width-1), 1.0f, 0.9f);
    }
}

////////////////////////////////////////////////////////////
// hue_background
//
// Create the initial background image: a hue ramp.
//
void
CMyD3DApplication::hue_background()
{
    m_tile_width = m_tile_height = 256;

    // create an image surface
    m_system_format = D3DFMT_A8R8G8B8;
    THR(m_pd3dDevice->CreateOffscreenPlainSurface(
        m_tile_width, m_tile_height, m_system_format,
        D3DPOOL_SYSTEMMEM, &m_system_tile, NULL));

    // lock the surface to initialize it
#if defined(RT_SURFACE_H)
    {
        rt::surface_lock lock(m_system_tile);

        // create one scanline of the surface
        D3DCOLOR *scanline = lock.scanline32(0);
        hue_ramp(scanline, m_tile_width);

        // initialize it using a smart lock
        for (UINT i = 1; i < m_tile_height; i++)
        {
            ::CopyMemory(lock.scanline32(i), scanline,
                m_tile_width*sizeof(D3DCOLOR));
        }
    }
#else
    {
        // initialize it using manual locking
        D3DLOCKED_RECT lr;
        THR(m_system_tile->LockRect(&lr, NULL, 0));

        D3DCOLOR *scanline =
            static_cast<D3DCOLOR *>(lr.pBits);
        hue_ramp(scanline, m_tile_width);

        BYTE *dest =
            static_cast<BYTE *>(lr.pBits) + lr.Pitch;
        for (UINT i = 1; i < m_tile_height; i++)
        {
            // replicate scanline across entire surface
            ::CopyMemory(dest, scanline,
                m_tile_width*sizeof(D3DCOLOR));
            dest += lr.Pitch;
        }
        THR(m_system_tile->UnlockRect());
    }
#endif
}
```

```

////////////////////////////////////////////////////////////
// init_background
//
// Create the system memory version of the background image.
//
void
CMyD3DApplication::init_background()
{
    switch (m_background)
    {
    case BACKGROUND_HUE_RAMP:
        hue_background();
        break;

    case BACKGROUND_IMAGE:
        {
            D3DXIMAGE_INFO info;
            THR(::D3DXGetImageInfoFromFile(
                m_background_file.c_str(), &info));
            m_tile_width = info.Width;
            m_tile_height = info.Height;
            m_system_format = D3DFMT_A8R8G8B8;
            THR(m_pd3dDevice->CreateOffscreenPlainSurface(
                m_tile_width, m_tile_height, m_system_format,
                D3DPOOL_SYSTEMMEM, &m_system_tile, NULL));
            THR(::D3DXLoadSurfaceFromFile(m_system_tile, NULL,
                NULL, m_background_file.c_str(), NULL,
                D3DX_FILTER_NONE, 0, NULL));
        }
        break;

    case BACKGROUND_GDI_ELLIPSE:
        m_tile_width = m_d3dsdBackBuffer.Width/3;
        m_tile_height = m_d3dsdBackBuffer.Height/3;
        m_system_format = D3DFMT_X8R8G8B8;
        break;

    default:
        ATLASSERT(false);
    }
}

////////////////////////////////////////////////////////////
// restore_background
//
```

```
// Restore the background image from system memory to
// device memory
//
void
CMyD3DApplication::restore_background()
{
    if (BACKGROUND_GDI_ELLIPSE == m_background)
    {
        m_tile_width = m_d3dsdBackBuffer.Width/3;
        m_tile_height = m_d3dsdBackBuffer.Height/3;
        THR(m_pd3dDevice->CreateOffscreenPlainSurface(
            m_tile_width, m_tile_height, m_system_format,
            D3DPOOL_SYSTEMMEM, &m_system_tile, NULL));

        // acquire the surface's DC via GetDC.  Control the
        // lifetime of the DC by the lifetime of the 'dc'
        // variable.
        {
            rt::c_surface_dc dc(m_system_tile);
            RECT r = { 0, 0, m_tile_width, m_tile_height };
            HBRUSH brush = static_cast<HBRUSH>(TWS(::GetStockObject(BLACK_BRUSH)
            TWS(::FillRect(dc, &r, brush));
            brush = static_cast<HBRUSH>(TWS(::GetStockObject(WHITE_BRUSH)));
            rt::c_push_gdi<HBRUSH> push(dc, brush);
            TWS(::Ellipse(dc, 20, 20, m_tile_width-20, m_tile_height-20));
        }
    }

    // create a tile surface in the back buffer format
    THR(m_pd3dDevice->CreateOffscreenPlainSurface(
        m_tile_width, m_tile_height, m_d3dsdBackBuffer.Format,
        D3DPOOL_DEFAULT, &m_device_tile, NULL));

    // the surface we built in InitDeviceObjects is A8R8G8B8,
    // but the back buffer may be a different format
    if (m_system_format == m_d3dsdBackBuffer.Format)
    {
        // we can copy the system surface directly
        THR(m_pd3dDevice->UpdateSurface(m_system_tile, NULL,
            m_device_tile, NULL));
    }
    else
    {
        // use D3DX to do the format conversion
        THR(::D3DXLoadSurfaceFromSurface(m_device_tile, NULL,
            NULL, m_system_tile, NULL, NULL, D3DX_FILTER_NONE,
            0));
    }
}

////////////////////////////////////////////////////////////
// recreate_background
//
// Destroy the existing background images and recreate them
//
void
CMyD3DApplication::recreate_background()
{
    m_system_tile = 0;
    m_device_tile = 0;
    init_background();
    restore_background();
}

bool
CMyD3DApplication::get_sprite_filename()
{
    rt::pauser pause(*this);
    TCHAR buffer[MAX_PATH] = { 0 };
    OPENFILENAME ofn =
    {
        sizeof(ofn), m_hWnd, NULL,
        _T("All image files\0")
            _T("*.bmp;*.dib;*.jpg;*.jpeg;*.png;")
            _T("*.dds;*.tga;*.pbm;*.pgm;*.ppm;*.pnm\0")
        _T("Bitmap images (*.bmp,*.dib)\0")
            _T("*.bmp;*.dib\0")
        _T("JPEG images (*.jpg,*.jpeg)\0")
            _T("*.jpg;*.jpeg\0")
        _T("PNG images (*.png)\0")
            _T("*.png\0")
        _T("DDS images (*.dds)\0")
            _T("*.dds\0")
        _T("Targa images (*.tga)\0")
            _T("*.tga\0")
        _T("PNM images (*.p[bgpn]m)\0")
            _T("*.pbm;*.pgm;*.ppm;*.pnm\0")
        _T("All files (*.*)\0")
            _T("*.*\0")
        _T("\0"), NULL, 0, 1, buffer, NUM_OF(buffer),
        NULL, 0, NULL, NULL,
        OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST
    };
    if (!m_dialogs && !m_bWindowed)
    {
        THR(ForceWindowed());
    }
    if (::GetOpenFileName(&ofn))
    {
        m_sprite_file = buffer;
        return true;
    }
    return false;
}

void
CMyD3DApplication::init_sprite()
{
    if (m_sprite_file.length() > 0)
    {
        THR(::D3DXCreateTextureFromFile(m_pd3dDevice,
            m_sprite_file.c_str(), &m_sprite_texture));
        D3DSURFACE_DESC sd;
        THR(m_sprite_texture->GetLevelDesc(0, &sd));
        const float scale = 64.0f/std::max(sd.Width, sd.Height);
        m_sprite_xform = rt::mat_scale(scale);//*rt::mat_trans(-32.0, -32.0, 0.0
    }
}

DWORD
CMyD3DApplication::PresentFlags() const
{
    return D3DPRESENTFLAG_LOCKABLE_BACKBUFFER | CD3DApplication::PresentFlags();
}
```