DirectDraw

This section provides information about the DirectDraw component. Information is divided into the following groups:

- About DirectDraw
- Why Use DirectDraw?
- Getting Started-Basic Graphics Concepts
- DirectDraw Architecture
- DirectDraw Essentials
- DirectDraw Tutorials
- DirectDraw Reference

About DirectDraw

DirectDraw® is a DirectX® SDK component that allows you to directly manipulate display memory, the hardware blitter, hardware overlay support, and flipping surface support. DirectDraw provides this functionality while maintaining compatibility with existing Microsoft® Windows®-based applications and device drivers.

DirectDraw is a software interface that provides direct access to display devices while maintaining compatibility with the Windows graphics device interface (GDI). It is not a high-level application programming interface (API) for graphics. DirectDraw provides a device-independent way for games and Windows subsystem software, such as 3-D graphics packages and digital video codecs, to gain access to the features of specific display devices.

DirectDraw works with a wide variety of display hardware, ranging from simple SVGA monitors to advanced hardware implementations that provide clipping, stretching, and non-RGB color format support. The interface is designed so that your applications can enumerate the capabilities of the underlying hardware and then use any supported hardware-accelerated features. Features that are not implemented in hardware are emulated by DirectX.

DirectDraw provides device-dependent access to display memory in a device-independent way. Essentially, DirectDraw manages display memory. Your application need only recognize some basic device dependencies that are standard across hardware implementations, such as RGB and YUV color formats and the pitch between raster lines. You need not call specific procedures to use the blitter or manipulate palette registers. Using DirectDraw, you can manipulate display memory with ease, taking full advantage of the blitting and color decompression capabilities of different types of display hardware without becoming dependent on a particular piece of hardware.

DirectDraw provides world-class game graphics on computers running Windows 95 and Windows NT® version 4.0 or later.

Why Use DirectDraw?

The DirectDraw component brings many powerful features to you, the Windows graphics programmer:

- The Hardware Abstraction Layer (HAL) of DirectDraw provides a consistent interface through which to work directly with the display and video memory, getting maximum performance from the system hardware.
- DirectDraw assesses the video hardware's capabilities, making use of special hardware features whenever possible. For example, if your video card supports hardware blitting, DirectDraw delegates blits to the video card, greatly increasing performance. Additionally, DirectDraw provides a Hardware Emulation Layer (HEL) to support features when the hardware does not.
- DirectDraw exists over Windows 95, gaining the advantage of 32-bit memory addressing and a flat memory model that the operating system provides. DirectDraw presents video and system memory as large blocks of storage, not as small segments. If you've ever used segment:offset addressing, you will quickly begin to appreciate this "flat" memory model.
- DirectDraw makes it easy for you to implement page flipping with multiple back buffers in full-screen applications. For more information, see Page Flipping and Back Buffering.
- Support for clipping in windowed or full-screen applications.
- Support for 3-D z-buffers.
- Support for hardware-assisted overlays with z-ordering.
- Access to image-stretching hardware.
- Simultaneous access to standard and enhanced display-device memory areas.
- Other features include custom and dynamic palettes, exclusive hardware access, and resolution switching.

These features combine to make it possible for you to write applications that easily out-perform standard Windows GDI-based applications and even MS-DOS applications.

Getting Started-Basic Graphics Concepts

This section provides an overview of graphics programming with DirectDraw. Each concept discussed here begins with a non-technical overview, followed by some specific information about how DirectDraw supports it.

To get the most from this overview, you don't need to be a graphics guru—in fact, if you are, you might want to skip this section entirely and move on to the more detailed information contained within the DirectDraw Essentials section. If you're familiar with Windows programming in C and C++, you won't have difficulty digesting this information. When you finish reading these topics, you will have a solid understanding of basic DirectDraw graphics programming concepts. The following topics are discussed:

- Device-Independent Bitmaps
- Drawing Surfaces
- Blitting Concepts
- Page Flipping and Back Buffering
- Introduction to Rectangles
- Sprite Concepts

Device-Independent Bitmaps

Windows, and therefore DirectX, uses the Device-Independent Bitmap (DIB) as its native graphics file format. Essentially, a DIB is a file that contains information describing an image's dimensions, the number of colors it uses, values describing those colors, and data that describes each pixel. Additionally, a DIB contains some lesser-used parameters, like information about file compression, significant colors (if all are not used), and physical dimensions of the image (in case it will end up in print). DIB files usually have the ".bmp" file extension, although they might occasionally have a ".dib" extension.

Because the DIB is so pervasive in Windows programming, the Platform SDK already contains many functions that you can use with DirectX. For example, the following application-defined function, taken from the ddutil.cpp file that comes with the DirectX APIs in the Platform SDK, combines Win32® and DirectX functions to load a DIB onto a DirectX surface.

```
extern "C" IDirectDrawSurface * DDLoadBitmap(IDirectDraw *pdd,
    LPCSTR szBitmap, int dx, int dy)
{
    HBITMAP             hbm;
    BITMAP              bm;
    DDSURFACEDESC       ddsd;
```

```
    IDirectDrawSurface *pdds;

    //
    //  This is the Win32 part.
    //   Try to load the bitmap as a resource, if that fails, try it as a file.
   //
    hbm = (HBITMAP)LoadImage(GetModuleHandle(NULL), szBitmap, IMAGE_BITMAP, dx, dy,
LR_CREATEDIBSECTION);

    if (hbm == NULL)
        hbm = (HBITMAP)LoadImage(NULL, szBitmap, IMAGE_BITMAP, dx, dy,
LR_LOADFROMFILE|LR_CREATEDIBSECTION);

    if (hbm == NULL)
        return NULL;

    //
    // Get the size of the bitmap.
    //
    GetObject(hbm, sizeof(bm), &bm);

    //
    // Now, return to DirectX function calls.
    // Create a DirectDrawSurface for this bitmap.
    //
    ZeroMemory(&ddsd, sizeof(ddsd));
    ddsd.dwSize = sizeof(ddsd);
    ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT |DDSD_WIDTH;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
    ddsd.dwWidth = bm.bmWidth;
    ddsd.dwHeight = bm.bmHeight;

    if (pdd->CreateSurface(&ddsd, &pdds, NULL) != DD_OK)
        return NULL;

    DDCopyBitmap(pdds, hbm, 0, 0, 0, 0);

    DeleteObject(hbm);

    return pdds;
}
```

For more detailed information about DIB files, see the Platform SDK.

Drawing Surfaces

Drawing surfaces receive video data to eventually be displayed on screen as images (bitmaps, to be exact). In most Windows programs, you get access to the drawing surface using a Win32 function such as **GetDC**, which stands for get the device context (DC). After you have the device context, you can start painting the screen. However, Win32 graphics functions are provided by an entirely different part of the system, the graphics device interface (GDI). The GDI is a system component that provides an abstraction layer that enables standard Windows applications to draw to the screen.

The drawback of GDI is that it wasn't designed for high-performance multimedia software, it was made to be used by business applications like word processors and spreadsheet applications. GDI provides access to a video buffer in system memory, not video memory, and doesn't take advantage of special features that some video cards provide. In short, GDI is great for most types of business software, but its performance is too slow for multimedia or game software.

On the other hand, DirectDraw can give you drawing surfaces that represent actual video memory. This means that when you use DirectDraw, you can write directly to the memory on the video card, making your graphics routines extremely fast. These surfaces are represented as contiguous blocks of memory, making it easy to perform addressing within them.

For more detailed information, see Surfaces.

Blitting Concepts

The term *blit* is shorthand for "bit block transfer," which is the process of transferring blocks of data from one place in memory to another. Graphics programmers use blitting to transfer graphics from one place in memory to another. Blits are often used to perform sprite animation, which is discussed later. For more information see, Sprite Concepts.

You can use the **IDirectDrawSurface3::Blt** and **IDirectDrawSurface3::BltFast** methods to perform blitting.

Page Flipping and Back Buffering

Page flipping is key in multimedia, animation, and game software. Software page flipping is analogous to the way cartoon artists animate their images. For example, the artist draws a figure on a sheet of paper, then sets it aside to work on the next frame. With each frame, the artist changes the figure slightly, so that when you flip between sheets rapidly the figure appears animated.

Page flipping in software is very similar to this process. Initially, you set up a series of DirectDraw surfaces that are designed to "flip" to the screen the way artist's paper flips to the next page. The first surface is referred to as the primary surface, and the

surfaces behind it are called back buffers . Your application writes to a back buffer, then flips the primary surface so that the back buffer appears on screen. While the system is displaying the image, your software is again writing to a back buffer. The process continues as long as you're animating, allowing you to animate images quickly and efficiently.

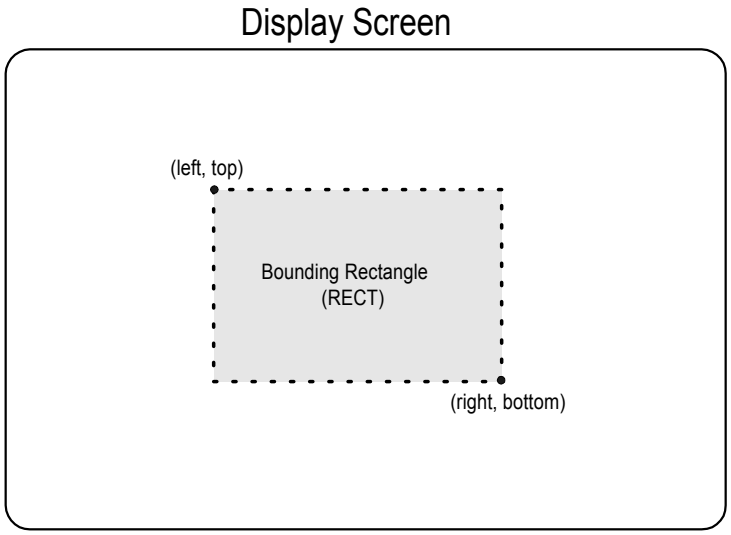
DirectDraw makes it easy for you to set up page flipping schemes, from a relatively simple double-buffered scheme (a primary surface with one back buffer) to more sophisticated schemes that add additional back buffers. For more information see DirectDraw Tutorials and Flipping Surfaces.

Introduction to Rectangles

Throughout DirectDraw and Windows programming, objects on the screen are referred to in terms of bounding rectangles. A bounding rectangle is described by two points, the top-left corner and bottom-right corner. Most applications use the **RECT** structure to carry information about a bounding rectangle to use when blitting to the screen or performing hit detection. The **RECT** structure has the following definition:

```
typedef struct tagRECT {
    LONG    left;    // This is the top-left corner's X-coordinate.
    LONG    top;     // The top-left corner's Y-coordinate.
    LONG    right;   // The bottom-right corner's X-coordinate.
    LONG    bottom;  // The bottom-right corner's Y-coordinate.
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;
```

In the preceding example, the **left** and **top** members are the X- and Y-coordinates of a bounding rectangle's top-left corner. Similarly, the **right** and **bottom** members make up the coordinates of the bottom-right corner. The following diagram illustrates how you can visualize these values:



Sprite Concepts

This section contains information about the basic concepts behind a common type of sprite animation. The following topics are discussed:

- What is a Sprite?
- Transparent Blitting and Color Keys
- Sprite and Patch Rectangles
- Bounds Checking and Hit Detection

What is a Sprite?

Many video games use sprites. In its most basic sense, a sprite is an image that moves around on the screen. The sprite is drawn onto a surface over the existing background, the composed scene is sent to the screen, then the sprite is redrawn in a new location, and the process repeats. Combine this with repairing the sprite's old location on the background and page flipping and you get the illusion that a sprite is moving around the screen.

Transparent Blitting and Color Keys

One challenge in sprite animation is accommodating nonrectangular sprites, which means almost all of them. Because blitting functions work with rectangles (for efficiency, consistency, and ease of use), your sprites must fit into rectangles as well, whether or not they actually look rectangular on the screen.

Although the concept might be confusing at first, this is how it works: The sprite image itself is nonrectangular, but is contained in a rectangular space where every pixel that is not part of the sprite is treated as "transparent" when the blitter is moving the image to its destination. The artist creating the sprite chooses an arbitrary color that will be used as the transparency "color key." This is typically a single uncommon color that the artist doesn't use for anything but transparency, but it can also be a specified range of colors.

Using the **IDirectDrawSurface3::SetColorKey** method, you can set the color key for a surface. After the color key is set, subsequent **IDirectDrawSurface3::BltFast** method calls can take advantage of that color key, ignoring the pixels that match it. This type of color key is known as a source color key. Because the source color key prevents “transparent” pixels from being written to the destination, the original background pixels are preserved in these places, making it look like the sprite is non-rectangular object and passing over the background.

Additionally, you can use a color key that affects the destination surface (a destination color key). A destination color key is a color on a surface that is used for pixels that can be overwritten by a sprite. In this case, for example, the artist might be working on a foreground image that sprites are supposed to pass behind, creating a layered effect. Again, the artist chooses an arbitrary color that isn't used elsewhere in the image, reserving it as a portion of the image where you are allowed to blit. When you blit a sprite to the destination surface with a destination color key specified, the sprite's pixels will only be blitted to pixels on the destination that are using the destination color key. Because the normal destination pixels are preserved, it looks like the sprite passes behind the image on the destination surface.

Sprite and Patch Rectangles

To complete the illusion of sprite movement, you need a way to erase the sprite's image from the background before you draw it at its new location. You could reload the entire background and redraw the sprite, but a great deal of performance would be lost. Instead, you can keep track of the rectangle that is the sprite's last location and redraw only that portion. This method is called "patching." To patch the sprite's old location, redraw the sprite's old location with a copy of the original background image, which you previously loaded on an off-screen surface. The process works well, because it doesn't waste a lot of processing time blitting an entire surface each cycle.

This process can be described in the following simple steps:

1. Set the patch rectangle to the last sprite location.
2. Patch the background at that location by blitting to the background image from the off-screen master copy.
3. Update the sprite's destination rectangle to reflect its new location.
4. Blit the sprite to its newly updated rectangle in the background image.
5. Repeat.

Using straightforward C/C++ combined with the graphics power provided by DirectDraw, you can implement this process to make a simple sprite engine.

Bounds Checking and Hit Detection

Bounds checking and hit detection are two very common and important tasks associated with sprites. Bounds checking is a term used to describe the process of limiting a sprite's possible range of motion. For example, you might want to limit a given sprite to keep it from moving off the screen. To do so, you can check the values for the sprite's location, which you'll probably keep in a **RECT** structure, and prevent them from changing beyond the limits of the screen resolution. DirectDraw doesn't provide bounds checking services, but you can easily implement a bounds checking scheme in C/C++ alone. Clipping, on the other hand, is supported by DirectDraw. For more information, see Clippers.

Hit detection, or collision detection, refers to the process of checking whether one or more sprites occupy the same place. Most hit detection schemes involve checking to see if the bounding rectangles for one or more sprites overlap. Because there are so many different types of hit detection schemes with an equally varied number of uses, DirectDraw doesn't support them for you, thereby giving you the freedom to implement a hit detection scheme that meets your application's needs.

DirectDraw Architecture

This section contains general information about the relationship between the DirectDraw component and the rest of DirectX, the operating system, and the system hardware. The following topics are discussed:

- Architectural Overview
- DirectDraw Object Types
- Hardware Abstraction Layer (HAL)
- Software Emulation

Architectural Overview

Multimedia software requires high-performance graphics. Through DirectDraw, Microsoft enables a much higher level of efficiency and speed in graphics-intensive applications for Windows than is possible with GDI, while maintaining device independence. DirectDraw provides tools to perform such key tasks as:

- Manipulating multiple display surfaces
- Accessing the video memory directly
- Page flipping
- Back buffering

- Managing the palette
- Clipping

Additionally, DirectDraw enables you to query the display hardware's capabilities at run time, then provide the best performance possible given the host computer's hardware capabilities.

As with other DirectX components, DirectDraw uses the hardware to its greatest advantage whenever possible, and provides software emulation for most features when hardware support is unavailable. Device independence is possible through use of the hardware-abstraction layer, or HAL. For more information about the HAL, see Hardware Abstraction Layer (HAL).

The DirectDraw component provides services through COM-based interfaces. In the most recent iteration, these interfaces are **IDirectDraw2**, **IDirectDrawSurface3**, **IDirectDrawPalette**, **IDirectDrawClipper**, and **IDirectDrawVideoPort**. Note that, in addition to these interfaces, DirectDraw continues to support all previous versions. For more information about COM concepts that you should understand to create applications with the DirectX APIs in the Platform SDK, see DirectX and the Component Object Model .

The DirectDraw object represents the display adapter and exposes its methods through the **IDirectDraw** and **IDirectDraw2** interfaces. In most cases you will use the **DirectDrawCreate** function to a DirectDraw object, but you can also create one with the **CoCreateInstance** COM function. For more information, see Creating DirectDraw Objects by Using CoCreateInstance.

After creating a DirectDraw object, you can create surfaces for it by calling the **IDirectDraw2::CreateSurface** method. Surfaces represent the memory on the display hardware, but can exist on either video memory or system memory. DirectDraw extends support for palettes, clipping (useful for windowed applications), and video ports through its other interfaces.

DirectDraw Object Types

You can think of DirectDraw as being composed of several objects that work together. This section briefly describes the objects you use when working with the DirectDraw component, organized by object type. For detailed information, see DirectDraw Essentials.

The DirectDraw component uses the following objects:

DirectDraw object

The DirectDraw object is the heart of all DirectDraw applications. It’s the first object you create, and you use it to make all other related objects. You create a DirectDraw object by calling the **DirectDrawCreate** function. DirectDraw objects expose their functionality through the **IDirectDraw** and **IDirectDraw2** interfaces. For more information, see The DirectDraw Object.

DirectDrawSurface object

The DirectDrawSurface object (casually referred to as a “surface”) represents an area in memory that holds data to be displayed on the monitor as images or moved to other surfaces. You can create a surface by calling the **IDirectDraw2::CreateSurface** method of the DirectDraw object with which it will be associated. DirectDrawSurface objects expose their functionality through the **IDirectDrawSurface**, **IDirectDrawSurface2**, and **IDirectDrawSurface3** interfaces. For more information, see Surfaces.

DirectDrawPalette object

The DirectDrawPalette object (casually referred to as a “palette”) represents a 16- or 256-color indexed palette to be used with a surface. It contains a series of indexed RGB triplets that describe colors associated with values within a surface. You do not use palettes with surfaces that use a pixel format depth greater that 8 bits. You can create a DirectDrawPalette object by calling the **IDirectDraw2::CreatePalette** method. DirectDrawPalette objects expose their functionality through the **IDirectDrawPalette** interface. For more information, see Palettes.

DirectDrawClipper object

The DirectDrawClipper object (casually referred to as a “clipper”) helps you prevent blitting to certain portions of a surface or beyond the bounds of a surface. You can create a clipper by calling the **IDirectDraw2::CreateClipper** method. DirectDrawClipper objects expose their functionality through the **IDirectDrawClipper** interface. For more information, see Clippers.

DirectDrawVideoPort object

The DirectDrawVideoPort object represents video-port hardware present in some systems. This hardware allows direct access to the frame buffer without accessing the CPU or using the PCI bus. You can create a DirectDrawVideoPort object by calling a **QueryInterface** method for the DirectDraw object, specifying the IID_IDDVideoPortContainer reference identifier. DirectDrawVideoPort objects expose their functionality through the **IDDVideoPortContainer** and **IDirectDrawVideoPort** interfaces. For more information, see Video Ports.

Hardware Abstraction Layer (HAL)

DirectDraw provides device independence through the hardware-abstraction layer (HAL). The HAL is a device-specific interface, provided by the device manufacturer, that DirectDraw uses to work directly with the display hardware. Applications never interact with the HAL. Rather, with the infrastructure that the HAL provides, DirectDraw exposes a consistent set of interfaces and methods that an application uses to display graphics. The device manufacturer implements the HAL in a combination of 16-bit and 32-bit code under Windows 95. Under Windows NT, the HAL is always implemented in 32-bit code. The HAL can be part of the display driver or a separate DLL that communicates with the display driver through a private interface that driver's creator defines.

The DirectDraw HAL is implemented by the chip manufacturer, board producer, or original equipment manufacturer (OEM). The HAL implements only device-

dependent code and performs no emulation. If a function is not performed by the hardware, the HAL does not report it as a hardware capability. Additionally, the HAL does not validate parameters; DirectDraw does this before the HAL is invoked.

Software Emulation

When the hardware does not support a feature through the hardware abstraction layer (HAL), DirectDraw attempts to emulate it. This emulated functionality is provided through the hardware-emulation layer (HEL). The HEL presents its capabilities to DirectDraw just as the HAL would. And, as with the HAL, applications never work directly with the HEL. The result is transparent support for almost all major features, regardless of whether a given feature is supported by hardware or through the HEL.

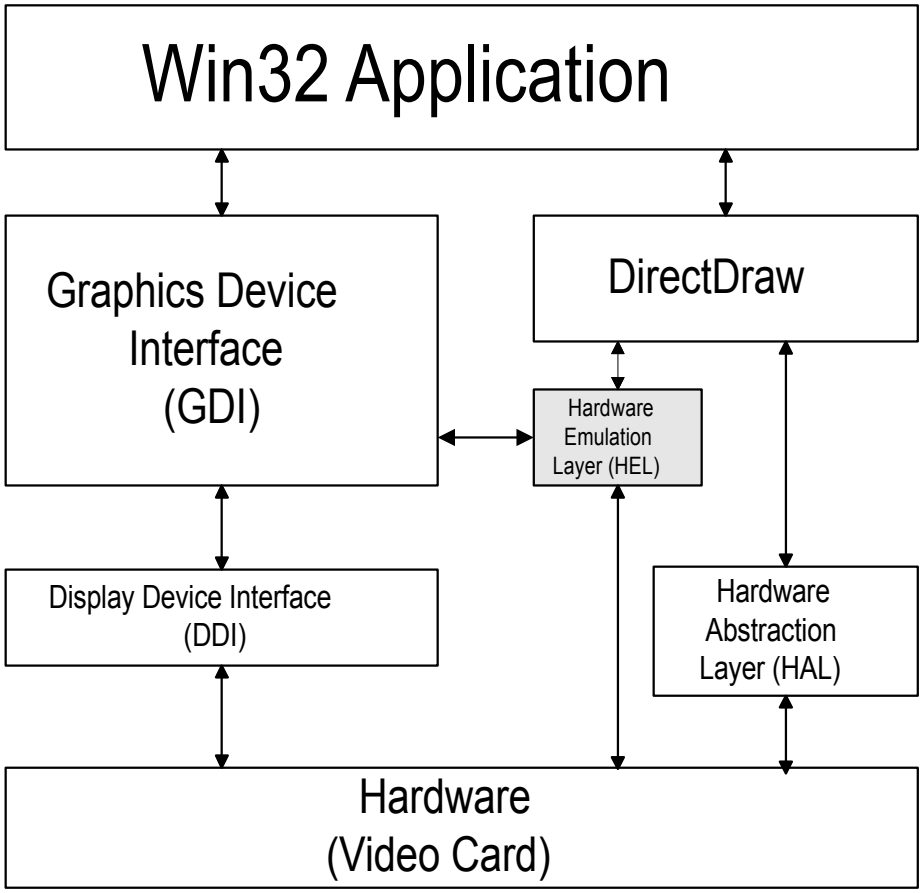
Obviously, software emulation cannot equal the performance that hardware features provide. You can query for the features the hardware supports by using the **IDirectDraw2::GetCaps** method. By examining these capabilities during application initialization, you can adjust application parameters to provide optimum performance over varying levels of hardware performance.

In some cases, certain combinations of hardware supported features and emulation can result in slower performance than emulation alone. For example, if the display device driver supports DirectDraw but not stretch blitting, noticeable performance losses will occur when stretch blitting from video memory surfaces. This happens because video memory is often slower than system memory, forcing the CPU to wait when accessing video memory surfaces. If your application uses a feature that isn't supported by the hardware, it is sometimes best to create surfaces in system memory, thereby avoiding performance losses created when the CPU accesses video memory.

For more information, see Hardware Abstraction Layer (HAL).

System Integration

The following diagram shows the relationships between DirectDraw, the graphics device interface (GDI), the hardware abstraction layer (HAL), hardware emulation layer (HEL) and the hardware.



As the preceding diagram shows, a DirectDraw object exists alongside GDI, and both have direct access to the hardware through a device-dependent abstraction layer. Unlike GDI, DirectDraw makes use of special hardware features whenever possible. If the hardware does not support a feature, DirectDraw attempts to emulate it by using the HEL. DirectDraw can provide surface memory in the form of a device context, making it possible for you to use GDI functions to work with surface objects.

DirectDraw Essentials

This section contains general information about the DirectDraw® component of DirectX®. Information is organized into the following groups:

- Cooperative Levels
- Display Modes
- The DirectDraw Object
- Surfaces

- Palettes
- Clippers
- Advanced DirectDraw Topics

Cooperative Levels

Cooperative levels describe how DirectDraw interacts with the display and how it reacts to events that might affect the display. Use the **IDirectDraw2::SetCooperativeLevel** method to set cooperative level of DirectDraw. For the most part, you use DirectDraw cooperative levels to determine whether your application runs as a full screen program with exclusive access to the display or as a windowed application. However, DirectDraw cooperative levels can also have the following effects:

- Enable DirectDraw to use Mode X resolutions. For more information, see Mode X and Mode 13 Display Modes.
- Prevent DirectDraw from releasing exclusive control of the display or rebooting if the user presses CTRL + ALT + DEL (exclusive mode only).
- Enable DirectDraw to minimize or maximize the application in response to activation events.

The normal cooperative level indicates that your DirectDraw application will operate as a windowed application. At this cooperative level you won't be able to change the primary surface's palette or perform page flipping. Additionally, you won’t be able to call some methods that drastically affect the display or video memory, such as **IDirectDraw2::Compact**.

At the full screen and exclusive cooperative level, you can use the hardware to its fullest. In this mode, you can set custom and dynamic palettes, change display resolutions, compact memory, and implement page flipping. The exclusive (full-screen) mode does not prevent other applications from allocating surfaces, nor does it exclude them from using DirectDraw or GDI. However, it does prevent applications other than the one currently with exclusive access from changing the display mode or palette.

Because applications can use DirectDraw with multiple windows, **IDirectDraw2::SetCooperativeLevel** does not require a window handle to be specified if the application is requesting the DDSCL_NORMAL mode. By passing a NULL to the window handle, all of the windows can be used simultaneously in normal Windows mode.

IDirectDraw2::SetCooperativeLevel maintains a binding between a process and a window handle. If **IDirectDraw2::SetCooperativeLevel** is called once in a process, a binding is established between the process and the window. If it is called again in the same process with a different non-null window handle, it returns the DDERR_HWNDALREADYSET error value. Some applications may receive this

error value when DirectSound® specifies a different window handle than DirectDraw —they should specify the same, top-level application window handle.

Display Modes

This section contains general information about DirectDraw display modes. The following topics are discussed:

- About Display Modes
- Determining Supported Display Modes
- Setting Display Modes
- Restoring Display Modes
- Mode X and Mode 13 Display Modes
- Support for High Resolutions and True-Color Bit Depths

About Display Modes

A display mode is a hardware setting that describes the dimensions and bit-depth of graphics that the display hardware sends to the monitor from the primary surface. Display modes are described by their defining characteristics: width, height, and bit-depth. For instance, most display adapters can display graphics 640 pixels wide and 480 pixels tall, where each pixel is 8 bits of data. In shorthand, this display mode could be called 640-by-480-by-8 (640x480x8). As the dimensions of a display mode get larger or as the bit-depth increases, more display memory is required.

There are two types of display modes: palettized and non-palettized. For palettized display modes, each pixel is a value representing an index into an associated palette. The bit depth of the display mode determines the number of colors that can be in the palette. For instance, in an 8-bit palettized display mode, each pixel is a value from 0 to 255. In such a display mode, the palette can contain 256 entries.

Non-palettized display modes, as their name states, do not use palettes. The bit depth of a non-palettized display mode indicates the total number of bits that are used to describe a pixel.

The primary surface and any surfaces in the primary flipping chain match the display mode’s dimensions, bit depth and pixel format. For more information, see Pixel Formats.

Determining Supported Display Modes

Because display hardware varies, not all devices will support all display modes. To determine the display modes supported on a given system, call the **IDirectDraw2::EnumDisplayModes** method. By setting the appropriate values and flags, the **IDirectDraw2::EnumDisplayModes** method can list all supported display modes or confirm that a single display mode that you specify is supported. The

method's first parameter, *dwFlags*, controls extra options for the method; in most cases, you will set *dwFlags* to 0 to ignore extra options. The second parameter, *lpDDSurfaceDesc*, is the address of a **DDSURFACEDESC** structure that describes a given display mode to be confirmed; you'll usually set this parameter to NULL to request that all modes be listed. The third parameter, *lpContext*, is a pointer that you want DirectDraw to pass to your callback function; if you don't need any extra data in the callback function, use NULL here. Last, you set the *lpEnumModesCallback* parameter to the address of the callback function that DirectDraw will call for each supported mode.

The callback function you supply when calling **IDirectDraw2::EnumDisplayModes** must match the prototype for the **EnumModesCallback** function. For each display mode that the hardware supports, DirectDraw calls your callback function passing two parameters. The first parameter is the address of a **DDSURFACEDESC** structure that describes one supported display mode, and the second parameter is the address of the application-defined data you specified when calling **IDirectDraw2::EnumDisplayModes**, if any.

Examine the values in the **DDSURFACEDESC** structure to determine the display mode it describes. The key structure members are the **dwWidth**, **dwHeight**, and **ddpfPixelFormat** members. The **dwWidth** and **dwHeight** members describe the display mode's dimensions, and the **ddpfPixelFormat** member is a **DDPIXELFORMAT** structure that contains information about the mode's bit depth.

The **DDPIXELFORMAT** structure carries information describing the mode's bit depth and tells you whether or not the display mode uses a palette. If the **dwFlags** member contains the DDPF_PALETTEINDEXED1, DDPF_PALETTEINDEXED2, DDPF_PALETTEINDEXED4, or DDPF_PALETTEINDEXED8 flag, the display mode's bit depth is 1, 2, 4 or 8 bits, and each pixel is an index into an associated palette. If **dwFlags** contains DDPF_RGB, then the display mode is non-palettized and its bit depth is provided in the **dwRGBBitCount** member of the **DDPIXELFORMAT** structure.

Setting Display Modes

You can set the display mode by using the **IDirectDraw2::SetDisplayMode** method. The **SetDisplayMode** method accepts four parameters that describe the dimensions, bit depth, and refresh rate of the mode to be set. The method uses a fifth parameter to indicate special options for the given mode; this is currently only used to differentiate between Mode 13 and the Mode X 320x200x8 display mode.

Although you can specify the desired display mode's bit depth, you cannot specify the pixel format that the display hardware will use for that bit depth. To determine the RGB bit masks that the display hardware uses for the current bit depth, call **IDirectDraw2::GetDisplayMode** after setting the display mode. If the current display mode is not palettized, you can examine the mask values in the **dwRBitMask**, **dwGBitMask**, and **dwBBitMask** members to determine the correct red, green, and blue bits. For more information, see Pixel Format Masks.

Modes can be changed by more than one application as long as they are all sharing a display card. You can change the bit depth of the display mode only if your application has exclusive access to the DirectDraw object. All DirectDrawSurface objects lose surface memory and become inoperative when the mode is changed. A surface's memory must be reallocated by using the **IDirectDrawSurface3::Restore** method.

The DirectDraw exclusive (full-screen) mode does not bar other applications from allocating DirectDrawSurface objects, nor does it exclude them from using DirectDraw or GDI functionality. However, it does prevent applications other than the one that obtained exclusive access from changing the display mode or palette.

Restoring Display Modes

You can explicitly restore the display hardware to its original mode by calling the **IDirectDraw2::RestoreDisplayMode** method. If the display mode was set by calling **IDirectDraw2::SetDisplayMode** (rather than **IDirectDraw::SetDisplayMode**) and your application takes the exclusive cooperative level, the original display mode is reset automatically when you set the application's cooperative level back to normal. If you're using the **IDirectDraw** interface, you must always explicitly restore the display mode.

Mode X and Mode 13 Display Modes

DirectDraw supports both Mode 13 and Mode X display modes. Mode 13 is the linear unflippable 320x200 8 bits per pixel palettized mode known widely by its hexadecimal BIOS mode number: 13. For more information, see Mode 13 Support. *Mode X* is a hybrid display mode derived from the standard VGA Mode 13. This mode allows the use of up to 256 kilobytes (KB) of display memory (rather than the 64 KB allowed by Mode 13) by using the VGA display adapter's EGA multiple video plane system.

On Windows 95, DirectDraw provides two Mode X modes (320×200×8 and 320×240×8) for all display cards. Some cards also support linear low-resolution modes. In linear low-resolution modes, the primary surface can be locked and directly accessed. This is not possible in Mode X modes.

Mode X modes are available only if an application uses the DDSCL_ALLOWMODEX, DDSCL_FULLSCREEN, and DDSCL_EXCLUSIVE flags when calling the **IDirectDraw2::SetCooperativeLevel** method. If DDSCL_ALLOWMODEX is not specified, the **IDirectDraw2::EnumDisplayModes** method will not enumerate Mode X modes, and the **IDirectDraw2::SetDisplayMode** method will fail if a Mode X mode is requested.

Windows 95 and Windows NT do not support Mode X modes; therefore, when your application is in a Mode X mode, you cannot use the **IDirectDrawSurface3::Lock** or **IDirectDrawSurface3::Blt** methods to lock or blit to the primary surface. You also cannot use either the **IDirectDrawSurface3::GetDC** method on the primary surface,

or GDI with a screen DC. Mode X modes are indicated by the DDSCAPS_MODEX flag in the **DDSCAPS** structure, which is part of the **DDSURFACEDESC** structure returned by the **IDirectDrawSurface3::GetCaps** and **IDirectDraw2::EnumDisplayModes** methods.

Mode X modes and some linear low-resolution modes are not supported on Windows NT.

Support for High Resolutions and True-Color Bit Depths

DirectDraw supports all of the screen resolutions and depths supported by the display device driver. DirectDraw allows an application to change the mode to any one supported by the computer's display driver, including all supported 24- and 32-bpp (true-color) modes.

DirectDraw also supports HEL blitting in true-color surfaces. If the display device driver supports blitting at these resolutions, the hardware blitter will be used for display-memory-to-display-memory blits. Otherwise, the HEL will be used to perform the blits.

Window 95 and Windows NT allow you to specify the type of monitor being used. DirectDraw checks a list of known display modes against the display restrictions of the installed monitor. If DirectDraw determines that the requested mode is not compatible with the monitor, the call to the **IDirectDraw2::SetDisplayMode** method fails. Only modes that are supported on the installed monitor will be enumerated when you call the **IDirectDraw2::EnumDisplayModes** method.

The DirectDraw Object

This section contains information about DirectDraw objects and how you can manipulate them through their **IDirectDraw** or **IDirectDraw2** interfaces. The following topics are discussed:

- What Are DirectDraw Objects?
- What's New in IDirectDraw2?
- Cooperative Levels
- Display Modes
- Multiple DirectDraw Objects per Process
- Creating DirectDraw Objects by Using CoCreateInstance

What Are DirectDraw Objects?

The DirectDraw object is the heart of all DirectDraw applications and is an integral part of Direct3D® applications as well. It is the first object you create and, through it,

you create all other related objects. Typically, you create a DirectDraw object by calling the **DirectDrawCreate** function, which returns an **IDirectDraw** interface. If you want to work with a different iteration of the interface (such as **IDirectDraw2**) to take advantage of new features it provides, you can query for it. Note that you can create multiple DirectDraw objects, one for each display device installed in a system.

The DirectDraw object represents the display device and makes use of hardware acceleration if the display device for which it was created supports hardware acceleration. Each unique DirectDraw object can manipulate the display device and create surfaces, palettes, and clipper objects that are dependent on (or are, "connected to") the object that created them. For example, to create surfaces, you call the **IDirectDraw2::CreateSurface** method. Or, if you need a palette object to apply to a surface, call the **IDirectDraw2::CreatePalette** method. Additionally, the **IDirectDraw2** interface exposes similar methods to create clipper objects.

You can create more than one instance of a DirectDraw object at a time. The simplest example of this is using two monitors on a Windows 95 system. Although Windows 95 does not support dual monitors on its own, it is possible to write a DirectDraw HAL for each display device. The display device Windows 95 and GDI recognizes is the one that will be used when you create the instance of the default DirectDraw object. The display device that Windows 95 and GDI do not recognize can be addressed by another, independent DirectDraw object that must be created by using the second display device's globally unique identifier (GUID). This GUID can be obtained by using the **DirectDrawEnumerate** function.

The DirectDraw object manages all of the objects it creates. It controls the default palette (if the primary surface is in 8-bits-per-pixel mode), the default color key, and the hardware display mode. It tracks what resources have been allocated and what resources remain to be allocated.

What's New in IDirectDraw2?

This section details new features provided by the **IDirectDraw2** interface and describes how it behaves differently than its predecessor, **IDirectDraw**. The following topics are discussed:

- New Features in IDirectDraw2
- Cooperative Levels and Display Modes with IDirectDraw2
- Getting an IDirectDraw2 Interface

New Features in IDirectDraw2

The **IDirectDraw2** interface extends the IDirectDraw interface by adding the **IDirectDraw2::GetAvailableVidMem** method. This method enables you to query the display hardware for information about the status of its total available video memory and how much of that memory is available to be used by a surface of a given type.

Cooperative Levels and Display Modes with IDirectDraw2

The interaction between the **IDirectDraw2::SetCooperativeLevel** and **IDirectDraw2::SetDisplayMode** methods differs from that of their **IDirectDraw** counterparts.

If your application uses the **IDirectDraw** interface to set the full-screen exclusive cooperative level and change the display mode, the display mode will not be automatically restored when you return to the normal cooperative level-you have to call the **IDirectDraw::RestoreDisplayMode** method. However, if you use the **IDirectDraw2** interface, calling **RestoreDisplayMode** isn't necessary. However, the **IDirectDraw2::RestoreDisplayMode** method is supported for applications that want to explicitly restore the original display mode.

Getting an IDirectDraw2 Interface

The Component Object Model on which DirectX is built specifies that an object can provide new functionality can be added through new interfaces, without affecting backward compatibility. To this end, the **IDirectDraw2** interface supersedes the **IDirectDraw** interface. This new interface can be obtained by using the **IDirectDraw::QueryInterface** method, as shown in the following C++ example:

```
// Create an IDirectDraw2 interface.
LPDIRECTDRAW  lpDD;
LPDIRECTDRAW2 lpDD2;

ddrval = DirectDrawCreate(NULL, &lpDD, NULL);
if(ddrval != DD_OK)
   return;

ddrval = lpDD->SetCooperativeLevel(hwnd,
   DDSCL_NORMAL);
if(ddrval != DD_OK)
   return;

ddrval = lpDD->QueryInterface(IID_IDirectDraw2,
   (LPVOID *)&lpDD2);
if(ddrval != DD_OK)
   return;
```

The preceding example creates a DirectDraw object, then calls the **IUnknown::QueryInterface** method of the **IDirectDraw** interface it received to create an **IDirectDraw2** interface.

After getting an **IDirectDraw2** interface, you can begin calling its methods to take advantage of new features, performance improvements, and behavioral differences. Because some methods might change with the release of a new interface, mixing

methods from an interface and its replacement (between **IDirectDraw** and **IDirectDraw2**, for example) can cause unpredictable results.

Multiple DirectDraw Objects per Process

DirectDraw allows a process to call the **DirectDrawCreate** function as many times as necessary. A unique and independent interface to a unique and independent DirectDraw object is returned after each call. Each DirectDraw object can be used as desired; there are no dependencies between the objects. Each object behaves exactly as if it had been created by a unique process.

DirectDraw objects are independent of one another and the DirectDrawSurface, DirectDrawPalette, and DirectDrawClipper objects they create should not be used with other DirectDraw objects because they are automatically released when the parent DirectDraw object is destroyed. If they are used with another DirectDraw object, they might stop functioning if their parent object is destroyed, causing the remaining DirectDraw object to malfunction.

The exception is DirectDrawClipper objects created by using the **DirectDrawCreateClipper** function. These objects are independent of any particular DirectDraw object and can be used with one or more DirectDraw objects.

Creating DirectDraw Objects by Using CoCreateInstance

You can create a DirectDraw object by using the **CoCreateInstance** function and the **IDirectDraw2::Initialize** method rather than the **DirectDrawCreate** function. The following steps describe how to create the DirectDraw object:

1 Initialize COM at the start of your application by calling **CoInitialize** and specifying NULL.

```
if (FAILED(CoInitialize(NULL)))
   return FALSE;
```

2 Create the DirectDraw object by using **CoCreateInstance** and the **IDirectDraw2::Initialize** method.

```
ddrval = CoCreateInstance(&CLSID_DirectDraw,
   NULL, CLSCTX_ALL, &IID_IDirectDraw2, &lpdd);
if(!FAILED(ddrval))
   ddrval = IDirectDraw2_Initialize(lpdd, NULL);
```

In this call to **CoCreateInstance**, the first parameter, *CLSID_DirectDraw*, is the class identifier of the DirectDraw driver object class, the *IID_IDirectDraw2* parameter identifies the particular DirectDraw interface to be created, and the *lpdd* parameter points to the DirectDraw object that is retrieved. If the call is successful, this function returns an uninitialized object.

3 Before you use the DirectDraw object, you must call **IDirectDraw2::Initialize**. This method takes the driver GUID parameter that the **DirectDrawCreate** function typically uses (NULL in this case). After the DirectDraw object is initialized, you can use and release it as if it had been created by using the **DirectDrawCreate** function. If you do not call the **IDirectDraw2::Initialize** method before using one of the methods associated with the DirectDraw object, a DDERR_NOTINITIALIZED error will occur.

Before you close the application, shut down COM by using the **CoUninitialize** function.

```
CoUnitialize();
```

Surfaces

This section contains information about DirectDrawSurface objects. The following topics are discussed:

- Basic Concepts
- Creating Surfaces
- Flipping Surfaces
- Losing Surfaces
- Releasing Surfaces
- Updating Surface Characteristics
- Accessing the Frame-Buffer Directly
- Using Non-local Video Memory Surfaces
- Converting Color and Format
- Overlay Surfaces
- Blitting to Multiple Windows

Basic Concepts

This section contains information about the basic concepts associated with DirectDrawSurface objects. The following topics are discussed:

- What Are Surfaces?
- Surface Interfaces
- Width and Pitch
- Color Keying
- Pixel Formats

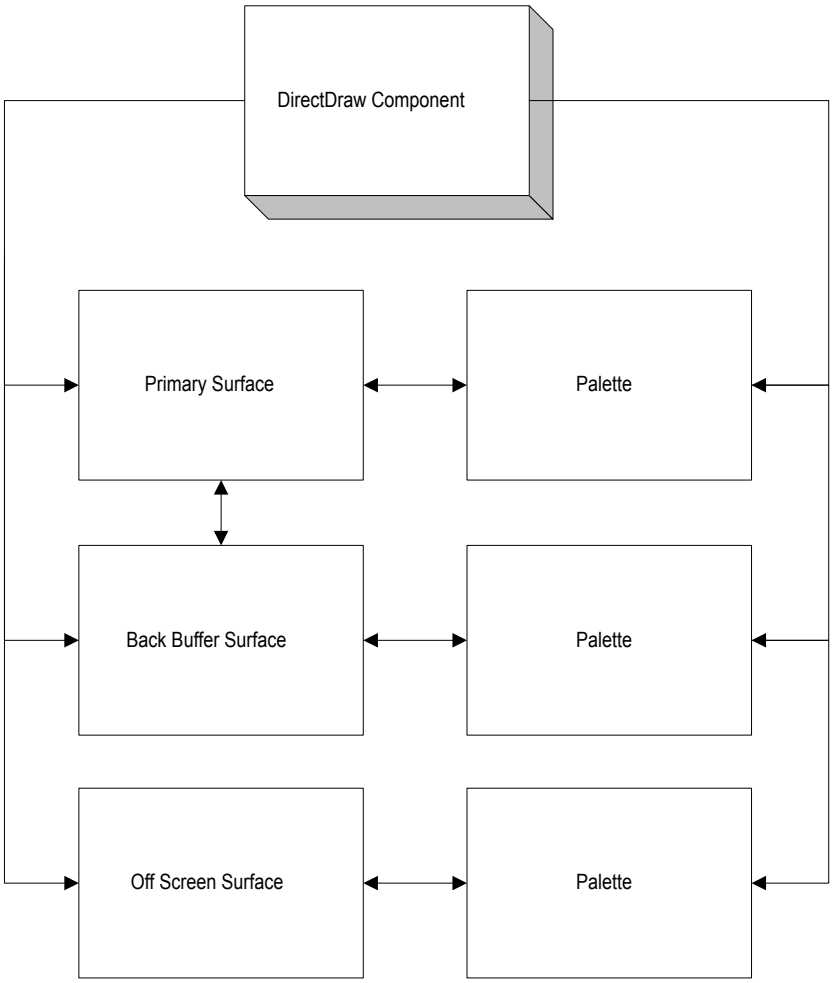
What Are Surfaces?

A surface, or DirectDrawSurface object, represents a linear area of display memory. A surface usually resides in the display memory of the display card, although surfaces can exist in system memory. Unless specifically instructed otherwise during the creation of the DirectDrawSurface object, DirectDraw object will put the DirectDrawSurface object wherever the best performance can be achieved given the requested capabilities. DirectDrawSurface objects can take advantage of specialized processors on display cards, not only to perform certain tasks faster, but to perform some tasks in parallel with the system CPU.

Using the **IDirectDraw2::CreateSurface** method, you can create a single surface object, complex surface-flipping chains, or three-dimensional surfaces. The **CreateSurface** method creates the requested surface or flipping chain and retrieves a pointer to the primary surface's **IDirectDrawSurface** interface through which the object exposes its functionality. If you want to work with a different iteration of the interface (such as **IDirectDrawSurface3**), you can query for it.

The **IDirectDrawSurface3** interface enables you to indirectly access memory through blit methods, such as **IDirectDrawSurface3::BltFast**. The surface object can provide a device context to the display that you can use with GDI functions. Additionally, you can use **IDirectDrawSurface3** methods to directly access display memory. For example, you can use the **IDirectDrawSurface3::Lock** method to lock the display memory and retrieve the address corresponding to that surface. Addresses of display memory might point to visible frame buffer memory (primary surface) or to nonvisible buffers (off-screen or overlay surfaces). Nonvisible buffers usually reside in display memory, but can be created in system memory if required by hardware limitations or if DirectDraw is performing software emulation. In addition, the **IDirectDrawSurface3** interface extends other methods that you can use to set or retrieve palettes, or to work with specific types or surfaces, like flipping chains or overlays.

From this illustration, you can see that all surface are created by a DirectDraw object and are often used closely with palettes. Although each surface object can be assigned a palette, palettes aren't required for anything but primary surfaces that use pixel formats of 8-bits in depth or less.



Surface Interfaces

DirectDrawSurface objects expose their functionality through the **IDirectDrawSurface**, **IDirectDrawSurface2**, and **IDirectDrawSurface3** interfaces. Each new interface version provides the same utility as its predecessors, with additional options available through new methods.

The **IDirectDrawSurface** interface is the oldest version of the interface and is provided by default when you create a surface by using the **IDirectDraw2::CreateSurface** method. To utilize the new functionality provided by another version of the interface, you must query for the new version by calling its **QueryInterface** method. The following example shows how you can do this:

```
LPDIRECTDRAWSURFACE  lpSurf;
LPDIRECTDRAWSURFACE2 lpSurf2;
```

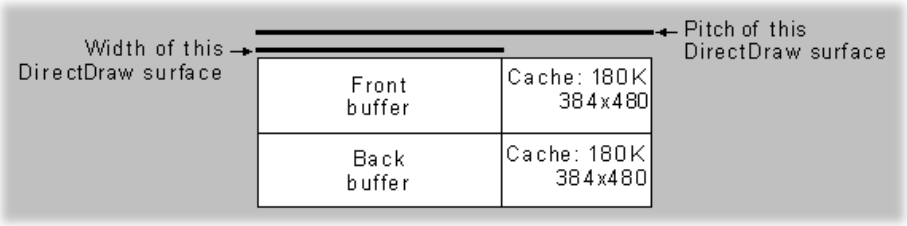
```
// Create surfaces.
memset(&ddsd, 0, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |
   DDSCAPS_SYSTEMMEMORY;
ddsd.dwWidth = 10;
ddsd.dwHeight = 10;

ddrval = lpDD2->CreateSurface(&ddsd, &lpSurf,
   NULL);
if(ddrval != DD_OK)
   return;

ddrval = lpSurf->QueryInterface(
   IID_IDirectDrawSurface2, (LPVOID *)&lpSurf2);
if(ddrval != DD_OK)
   return;

ddrval = lpSurf2->PageLock(0);
if(ddrval != DD_OK)
   return;

ddrval = lpSurf2->PageUnlock(0);
if(ddrval != DD_OK)
   return;
```

The preceding example retrieves a DirectDrawSurface object's **IDirectDrawSurface2** interface by specifying the IID_IDirectDraw2 reference identifier when it calls the **QueryInterface** method. To retrieve an **IDirectDrawSurface3** interface, use the IID_IDirectDrawSurface3 reference identifier instead.

Width and Pitch

If your application writes to display memory, bitmaps stored in memory do not necessarily occupy a contiguous block of memory. In this case, the *width* and *pitch* of a line in a bitmap can be different from each other. The width is the distance between two addresses in memory that represent the beginning of a line and the end of the line of a stored bitmap. This distance represents only the width of the bitmap in memory; it does not include any extra memory required to reach the beginning of the next line of the bitmap. The pitch is the distance between two addresses in memory that represent the beginning of a line and the beginning of the next line in a stored bitmap.

For rectangular memory, for example, the pitch of the display memory will include the width of the bitmap plus part of a cache. The following figure shows the difference between width and pitch in rectangular memory:



In this figure, the front buffer and back buffer are both 640×480×8, and the cache is 384×480×8. To reach the address of the next line to write to the buffer, you must add 640 and 384 to get 1024, which is the beginning of the next line.

Therefore, when rendering directly into surface memory, always use the pitch returned by the **IDirectDrawSurface3::Lock** method (or the **IDirectDrawSurface3::GetDC** method). Do not assume a pitch based solely on the display mode. If your application works on some display adapters but looks garbled on others, this may be the cause of your problem.

Color Keying

DirectDraw supports source and destination *color keying* for blits and overlay surfaces. You can supply a color key or a color range for both of these types of color keying. For general information about color keying, see Transparent Blitting and Color Keys. You set a surface's color key by calling the its **IDirectDrawSurface3::SetColorKey** method.

When blitting, *source color keying* specifies a color or color range that is not copied. Likewise, *destination color keying* specifies a color or color range that is replaced The source color key specifies what can and cannot be read from the surface. The destination color key specifies what can and cannot be written onto, or covered up, on the destination surface. If a destination surface has a color key, only the pixels that match the color key are changed, or covered up, on the destination surface.

In addition to blit-related color keys, overlay surfaces can use overlay color keys. For more information, see Overlay Color Keys.

Some hardware supports color ranges only for YUV pixel data. YUV data is usually video, and the transparent background may not be a single color due to quantitization errors during conversion. Content should be written to a single transparent color whenever possible, regardless of pixel format.

Color keys are specified using the pixel format of a surface. If a surface is in a palettized format, the color key is specified as an index or a range of indices. If the surface's pixel format is specified by a FOURCC code that describes a YUV format, the YUV color key is specified by the three low-order bytes in both the **dwColorSpaceLowValue** and **dwColorSpaceHighValue** members of the **DDCOLORKEY** structure. The lowest order byte contains the V data, the second lowest order byte contains the U data, and the highest order byte contains the Y data. The *dwFlags* parameter of the **IDirectDrawSurface3::SetColorKey** method

specifies whether the color key is to be used for overlay or blit operations, and whether it is a source or a destination key. Some examples of valid color keys follow:

8-bit palettized mode

```
// Palette entry 26 is the color key.
dwColorSpaceLowValue = 26;
dwColorSpaceHighValue = 26;
```

24-bit true-color mode

```
// Color 255,128,128 is the color key.
dwColorSpaceLowValue = RGBQUAD(255,128,128);
dwColorSpaceHighValue = RGBQUAD(255,128,128);
```

FourCC YUV mode

```
// Any YUV color where Y is between 100 and 110
// and U or V is between 50 and 55 is transparent.
dwColorSpaceLowValue = YUVQUAD(100,50,50);
dwColorSpaceHighValue = YUVQUAD(110,55,55);
```

Pixel Formats

Pixel formats dictate how data for each pixel in surface memory is to be interpreted. DirectDraw uses the **DDPIXELFORMAT** structure to describe various pixel formats. The **DDPIXELFORMAT** contains members to describe the following traits of a pixel format:

- Palettized or non-palettized pixel format
- If non-palettized, whether the pixel format is RGB or YUV
- Bit depth
- Bit masks for the pixel format's components

You can retrieve information about an existing surface's pixel format by calling the **IDirectDrawSurface3::GetPixelFormat** method.

Creating Surfaces

The DirectDrawSurface object represents a surface that usually resides in the display memory, but can exist in system memory if display memory is exhausted or if it is explicitly requested.

Use the **IDirectDraw2::CreateSurface** method to create one surface or to simultaneously create multiple surfaces (a complex surface). When calling **CreateSurface**, you specify the dimensions of the surface, whether it is a single surface or a complex surface, and the pixel format (if the surface won't be using an indexed palette). All these characteristics are contained in a **DDSURFACEDESC**

structure, whose address you send with the call. If the hardware can't support the requested capabilities or if it previously allocated those resources to another DirectDrawSurface object, the call will fail.

Creating single surfaces or multiple surfaces is a simple matter that requires only a few lines of code. There are four main scenarios for creating surfaces. Each scenario requires a little more preparation than the one before it, but none are difficult. The following four scenarios are discussed:

1. Creating the Primary Surface
2. Creating an Off-Screen Surface
3. Creating Complex Surfaces and Flipping Chains
4. Creating Wide Surfaces

By default, DirectDraw attempts to create a surface in local video memory. If there isn't enough local video memory available to hold the surface, DirectDraw will try to use non-local video memory (on some AGP-equipped systems), and fall back on system memory if all other types of memory are unavailable. You can explicitly request that a surface be created in a certain type of memory by including the appropriate flags in the associated **DDSCAPS** structure when calling **CreateSurface**.

Creating the Primary Surface

The primary surface is the surface currently visible on the monitor and is identified by the DDSCAPS_PRIMARYSURFACE flag. You can only have one primary surface for each DirectDraw object.

When you create a primary surface, the dimensions implicitly match the current display mode. Therefore, this is the one time you don't need to declare surface dimensions. Frankly, if you do specify them, the call fails–even if they match the current display mode.

The following example shows how to prepare the **DDSURFACEDESC** structure members relevant for creating the primary surface.

```
DDSURFACEDESC ddsd;
ddsd.dwSize = sizeof(ddsd);

// Tell DirectDraw which members are valid.
ddsd.dwFlags = DDSD_CAPS;

// Request a primary surface.
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
```

Creating an Off-Screen Surface

An off-screen surface is often used to cache bitmaps that will later be blitted to the primary surface or a back buffer. You must declare the dimensions of an off-screen

surface by including the DDSC_WIDTH and DDSD_HEIGHT flags and the corresponding values in the **dwWidth** and **dwHeight** members. Additionally, you must include the DDSCAPS_OFFSCREENPLAIN flag in the accompanying **DDSCAPS** structure.

By default, DirectDraw creates a surface in display memory unless it will not fit, in which case it creates the surface in system memory. You can explicitly choose display or system memory by including the DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY flags in the **dwCaps** member of the **DDSCAPS** structure. The method fails, returning an error, if it can't create the surface in the specified location.

The following example shows how to prepare for creating a simple off-screen surface.

```
DDSURFACEDESC ddsd;
ddsd.dwSize = sizeof(ddsd);

// Tell DirectDraw which members are valid.
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;

// Request a simple off-screen surface, sized
// 100 by 100 pixels.
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
dwHeight = 100;
dwWidth = 100;
```

In previous versions of DirectX, the maximum width of off-screen surfaces was limited to the width of the primary surface. With DirectX 5, you can create surfaces as wide as you need, permitting that the display hardware can support them. Be careful when declaring wide off-screen surfaces; if the video card memory cannot hold a surface as wide as you request, the surface is created in system memory. If you explicitly choose video memory and the hardware can't support it, the call fails.

Creating Complex Surfaces and Flipping Chains

You can also create complex surfaces. A complex surface is a set of surfaces created with a single call to the **IDirectDraw2::CreateSurface** method. If the DDSCAPS_COMPLEX flag is set when you call **CreateSurface** call, DirectDraw implicitly creates one or more surfaces in addition to the surface explicitly specified. You manage complex surfaces just like a single surface—a single call to the **IDirectDraw::Release** method releases all surfaces, and a single call to the **IDirectDrawSurface3::Restore** method restores them all. However, implicitly created surfaces cannot be detached. For more information, see **IDirectDrawSurface3::DeleteAttachedSurface**.

One of the most useful complex surfaces you can create is a flipping chain. Usually, a flipping chain is made of a primary surface and one or more back buffers. The DDSCAPS_FLIP flag indicates that a surface is part of a flipping chain. Creating a

flipping chain this way requires that you also include the DDSCAPS_COMPLEX flag.

The following example shows how to prepare for creating a primary surface flipping chain.

```
DDSURFACEDESC ddsd;
ddsd.dwSize = sizeof(ddsd);

// Tell DirectDraw which members are valid.
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;

// Request a primary surface with a single
// back buffer
ddsd.ddsCaps.dwCaps = DDSCAPS_COMPLEX | DDSCAPS_FLIP |
DDSCAPS_PRIMARYSURFACE;
ddsd.dwBackBufferCount = 1;
```

The previous example constructs a double-buffered flipping environment – a single call to the **IDirectDrawSurface3::Flip** method exchanges the surface memory of the primary surface and the back buffer. If you specify 2 for the value of the **dwBackBufferCount** member of the **DDSURFACEDESC** structure, two back buffers are created, and each call to **Flip** rotates the surfaces in a circular pattern, providing a triple-buffered flipping environment.

Creating Wide Surfaces

DirectDraw allows you to create off-screen surfaces in video memory that are wider that the primary surface. This is only possible when display device support for wide surfaces is present.

To check for wide surface support, call **IDirectDraw2::GetCaps** and look for the DDCAPS2_WIDESURFACES flag in the **dwCaps2** member of the first **DDCAPS** structure you send with the call. If the flag is present, you can create video memory off-screen surfaces that are wider that the primary surface.

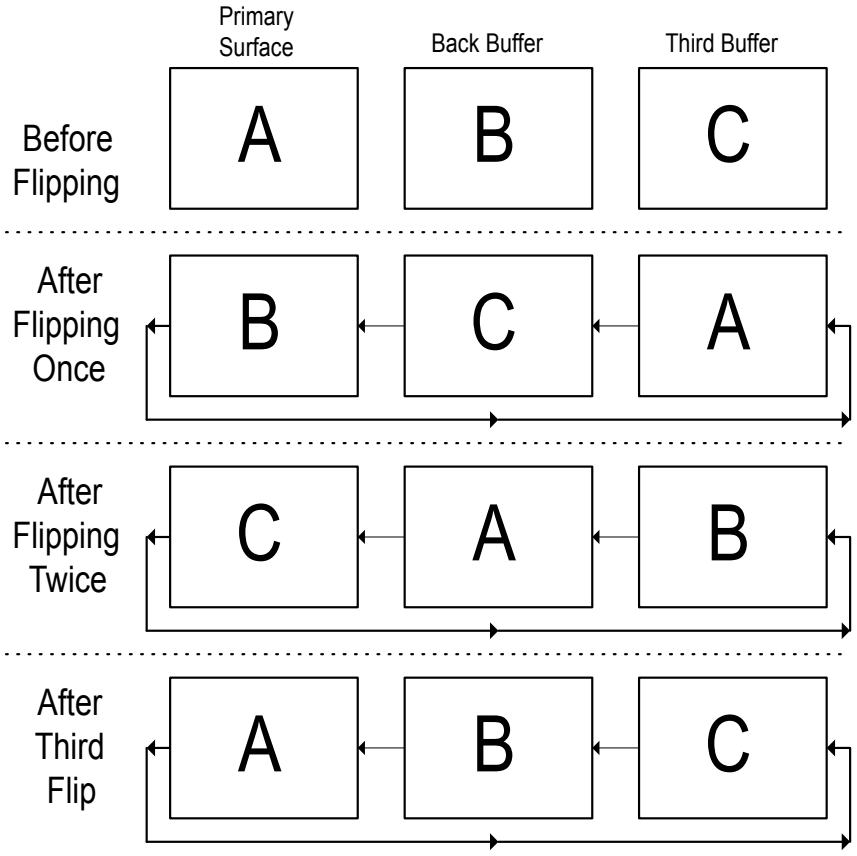
If you attempt to create a wide surface in video memory when the DDCAPS2_WIDESURFACES flag isn't present, the attempt will fail and return DDERR_INVALIDPARAMS.

Wide surfaces are always supported for system memory surfaces, video port surfaces, and execute buffers.

Flipping Surfaces

Any surface in DirectDraw can be constructed as a *flipping surface*. A flipping surface is any piece of memory that can be swapped between a *front buffer* and a *back buffer*. (this construct is commonly referred to as a *flipping chain*). Often, the front buffer is the primary surface, but it doesn't have to be.

Typically, when you use the **IDirectDrawSurface3::Flip** method to request a surface flip operation, the pointers to surface memory for the primary surface and back buffers are swapped. Flipping is performed by switching pointers that the display device uses for referencing memory, not by copying surface memory. (The exception to this is when DirectDraw is emulating the flip, in which case it simply copies the surfaces. DirectDraw emulates flip operations if a back buffer cannot fit into display memory or if the hardware doesn't support DirectDraw.) When a flipping chain contains a primary surface and more than one back-buffer, the pointers are switched in a circular pattern, as shown in the following illustration:



Other surfaces that are attached to a DirectDraw object, but not part of the flipping chain, are unaffected when the **Flip** method is called.

Remember, DirectDraw flips surfaces by swapping surface memory pointers within DirectDrawSurface objects, not by swapping the objects themselves. This means that, to blit to the back buffer in any type of flipping scheme, you always use the same DirectDrawSurface object-the one that was the back buffer when you created the flipping chain. Conversely, you always perform a flip operation by calling the front surface's **Flip** method.

When working with visible surfaces, such as a primary surface flipping chain or a visible overlay surface flipping chain, the **Flip** method is asynchronous unless you include the DDFLIP_WAIT flag. On these visible surfaces, the **Flip** method can return before the actual flip operation occurs in the hardware (because the hardware doesn't flip until the next vertical refresh occurs). While the actual flip operation is pending, the back buffer behind the currently visible surface can't be locked or blitted by calling the **IDirectDrawSurface3::Lock**, **IDirectDrawSurface3::Blt**, **IDirectDrawSurface3::BltFast**, or **IDirectDrawSurface3::GetDC** methods. If you attempt to call these methods while a flip operation is pending, they will fail and return DDERR_WASSTILLDRAWING. However, if you are using a triple buffered scheme, the rearmost buffer is still available.

Losing Surfaces

The surface memory associated with a DirectDrawSurface object may be freed, while the DirectDrawSurface objects representing these pieces of surface memory are not necessarily released. When a DirectDrawSurface object loses its surface memory, many methods return DDERR_SURFACELOST and perform no other action.

Surfaces can be lost because the display card mode was changed or because another application received exclusive access to the display card and freed all of the surface memory currently allocated on the card. The **IDirectDrawSurface3::Restore** method re-creates these lost surfaces and reconnects them to their DirectDrawSurface object. Restoring a surface doesn't reload any bitmaps that may have existed in the surface prior to its loss. Therefore, if you lose a surface you must also completely reconstitute the graphics it once held.

For more information, see Setting Display Modes.

Releasing Surfaces

Like all COM interfaces, you must release surfaces by calling the **Release** method when you no longer need them.

Each surface you individually create must be explicitly released. However, if you implicitly created multiple surfaces with a single call to **IDirectDraw2::CreateSurface** or **IDirectDraw::CreateSurface**, such as a flipping chain, you need only release the front buffer. In this case, any pointers you might have to back buffer surfaces are implicitly released and can no longer be used.

Updating Surface Characteristics

You can update the characteristics of an existing surface by using the **IDirectDrawSurface3::SetSurfaceDesc** method. With this method, you can change the pixel format and location of a DirectDrawSurface object's surface memory to system memory that your application has explicitly allocated. This is useful as it allows a surface to use data from a previously allocated buffer without copying. The new surface memory is allocated by the client application and, as such, the client

application must also deallocate it. For more information about how this method is used, see Updating Surface Characteristics.

When calling the **IDirectDrawSurface3::SetSurfaceDesc** method, the *lpddsd* parameter must be the address of a **DDSURFACEDESC** structure that describes the new surface memory as well as a pointer to that memory. Within the structure, you can only set the **dwFlags** member to reflect valid members for the location of the surface memory, dimensions, pitch, and pixel format. Therefore, **dwFlags** can only contain combinations of the DDSD_WIDTH, DDSD_HEIGHT, DDSD_PITCH, DDSD_LPSURFACE, and DDSD_PIXELFORMAT flags, which you set to indicate valid structure members.

Before you set the values in the structure, you must allocate memory to hold the surface. The size of the memory you allocate is important. Not only do you need to allocate enough memory to accommodate the surface's width and height, but you need to have enough to make room for the surface pitch, which must be a QWORD (8 byte) multiple. Remember, pitch is measured in bytes, not pixels.

When setting surface values in the structure, the **lpSurface** member is a pointer to the memory you allocated and the **dwHeight** and **dwWidth** members describe the surface dimensions in pixels. If you specify surface dimensions, you must fill the **lPitch** member to reflect the surface pitch as well. Pitch must be a DWORD multiple. Likewise, if you specify pitch, you must also specify a width value. Lastly, the **ddpfPixelFormat** member describes the pixel format for the surface. With the exception of the **lpSurface** member, if you don't specify a value for these members, the method defaults to using the value from the current surface.

There are some restrictions you must be aware of when using **IDirectDrawSurface3::SetSurfaceDesc**, some of which are common sense. For example, the **lpSurface** member of the **DDSURFACEDESC** structure must be a valid pointer to a system memory (the method doesn't support video memory pointers at this time). Also, the **dwWidth** and **dwHeight** members must be nonzero values. Lastly, you cannot reassign the primary surface or any surfaces within the primary's flipping chain.

You can set the same memory for multiple DirectDrawSurface objects, but you must take care that the memory is not deallocated while it is assigned to any surface object.

Using the **SetSurfaceDesc** method incorrectly will cause unpredictable behavior. The DirectDrawSurface object will not deallocate surface memory that it didn't allocate. Therefore, when the surface memory is no longer needed, it is your responsibility to deallocate it. However, when **SetSurfaceDesc** is called, DirectDraw frees the original surface memory that it implicitly allocated when creating the surface.

Accessing the Frame-Buffer Directly

You can directly access surface memory in the frame-buffer or in system memory by using the **IDirectDrawSurface3::Lock** method. When you call this method, the *lpDestRect* parameter is a pointer to a **RECT** structure that describes the rectangle on the surface you want to access directly. To request that the entire surface be locked,

set *lpDestRect* to NULL. Also, you can specify a **RECT** that covers only a portion of the surface. Providing that no two rectangles overlap, two threads or processes can simultaneously lock multiple rectangles in a surface.

The **Lock** method fills a **DDSURFACEDESC** structure with all the information you need to properly access the surface memory. The structure includes information about the pitch (or stride) and the pixel format of the surface, if different from the pixel format of the primary surface. When you finish accessing the surface memory, call the **IDirectDrawSurface3::Unlock** method to unlock it.

While you have a surface locked, you can directly manipulate the contents. The following list describes some tips for avoiding common problems with directly rendering surface memory:

- Never assume a constant display pitch. Always examine the pitch information returned by the **IDirectDrawSurface3::Lock** method. This pitch can vary for a number of reasons, including the location of the surface memory, the type of display card, or even the version of the DirectDraw driver. For more information, see Width and Pitch.
- Make certain you blit to unlocked surfaces. DirectDraw blit methods will fail, returning DDERR_SURFACEBUSY or DDERR_LOCKEDSURFACES, if called on a locked surface. Similarly, GDI blit functions fail without returning error values if called on a locked surface that exists in display memory.
- Limit your application's activity while a surface is locked. While a surface is locked, DirectDraw often holds the Win16Lock so that gaining access to surface memory can occur safely. The Win16Lock serializes access to GDI and USER, shutting down Windows for the duration between the **IDirectDrawSurface3::Lock** and **IDirectDrawSurface3::Unlock** calls. The **IDirectDrawSurface3::GetDC** method implicitly calls **IDirectDrawSurface3::Lock**, and the **IDirectDrawSurface3::ReleaseDC** implicitly calls **IDirectDrawSurface3::Unlock**.
- Copy aligned to display memory. Windows 95 uses a page fault handler, Vflatd.386, to implement a virtual flat-frame buffer for display cards with bank-switched memory. The handler allows these display devices to present a linear frame buffer to DirectDraw. Copying unaligned to display memory can cause the system to suspend operations if the copy spans memory banks.

Locking the surface typically causes DirectDraw to take the Win16Lock. During the Win16Lock all other applications, including Windows, cease execution. Since the Win16Lock stops applications from executing, standard debuggers cannot be used while the lock is held. Kernel debuggers can be used during this period.

If a blit is in progress when you call **IDirectDrawSurface3::Lock**, the method will return immediately with an error, as a lock cannot be obtained. To prevent the error, use the DDLOCK_WAIT flag to cause the method to wait until a lock can be successfully obtained.

Using Non-local Video Memory Surfaces

DirectDraw supports the Advanced Graphics Port (AGP) architecture for creating surfaces in non-local video memory. On AGP-equipped systems, DirectDraw will use non-local video memory if local video memory is exhausted or if non-local video memory is explicitly requested, depending on the type of AGP implentation that is in place.

Currently, there are two implementations of the AGP architecture, known as the "execute model" and the "DMA model." In the execute model implementation, the display device supports the same features for non-local video memory surfaces and local video memory surfaces. As a result, when you retrieve hardware capabilities by calling the **IDirectDraw2::GetCaps** method, the blit-related flags in the **dwNLVBCaps**, **dwNLVBCaps2**, **dwNLVBCKeyCaps**, **dwNLVBFXCaps**, and **dwNLVBRops** members of the **DDCAPS** structure will be identical to those for local video memory. Under the execute model, if local video memory is exhausted, DirectDraw will automatically fall back on non-local video memory unless the caller specifically requests otherwise.

In the DMA model implementation, support for blitting and texturing from non-local video memory surfaces is limited. When the display device uses the DMA model, the DDCAPS2_NONLOCALVIDMEMCAPS flag will be set in the **dwCaps2** member when you retrieve device capabilities. In the DMA model, the blit-related flags included in the **dwNLVBCaps**, **dwNLVBCaps2**, **dwNLVBCKeyCaps**, **dwNLVBFXCaps**, and **dwNLVBRops** members of the **DDCAPS** structure describe the features that are supported; these features will often be a smaller subset of those supported for local video memory surfaces. Under the DMA model, DirectDraw will never create a surface in non-local video memory unless the caller explicitly requests it.

DMA model implementations vary in support for texturing from non-local video memory surfaces. If the driver supports texturing from non-local video memory surfaces, the D3DDEVCAPS_TEXTURENONLOCALVIDMEM flag will be set when you retrieve the 3-D device's capabilities by calling the **IDirect3DDevice2::GetCaps** method.

Converting Color and Format

Non-RGB surface formats are described by four-character codes (FOURCC codes). If an application calls the **IDirectDrawSurface3::GetPixelFormat** method to request the pixel format, and the surface is a non-RGB surface, the DDPF_FOURCC flag will be set and the **dwFourCC** member of the **DDPIXELFORMAT** structure will be valid. If the FOURCC code represents a YUV format, the DDPF_YUV flag will also be set and the **dwYUVBitCount**, **dwYBits**, **dwUBits**, **dwVBits**, and **dwYUVAlphaBits** members will be valid masks that can be used to extract information from the pixels.

If an RGB format is present, the DDPF_RGB flag will be set and the **dwRGBBitCount**, **dwRBits**, **dwGBits**, **dwBBits**, and **dwRGBAlphaBits** members

will be valid masks that can be used to extract information from the pixels. The DDPF_RGB flag can be set in conjunction with the DDPF_FOURCC flag if a nonstandard RGB format is being described.

During color and format conversion, two sets of FOURCC codes are exposed to the application. One set of FOURCC codes represents the capabilities of the blitting hardware; the other represents the capabilities of the overlay hardware.

For more information, see Four Character Codes (FOURCC).

Overlay Surfaces

This section contains information about DirectDraw overlay surface support. The following topics are discussed:

- Overlay Surface Overview
- Significant **DDCAPS** Members and Flags
- Source and Destination Rectangles
- Boundary and Size Alignment
- Minimum and Maximum Stretch Factors
- Overlay Color Keys
- Positioning Overlay Surfaces
- Creating Overlay Surfaces
- Overlay Z-Orders
- Flipping Overlay Surfaces

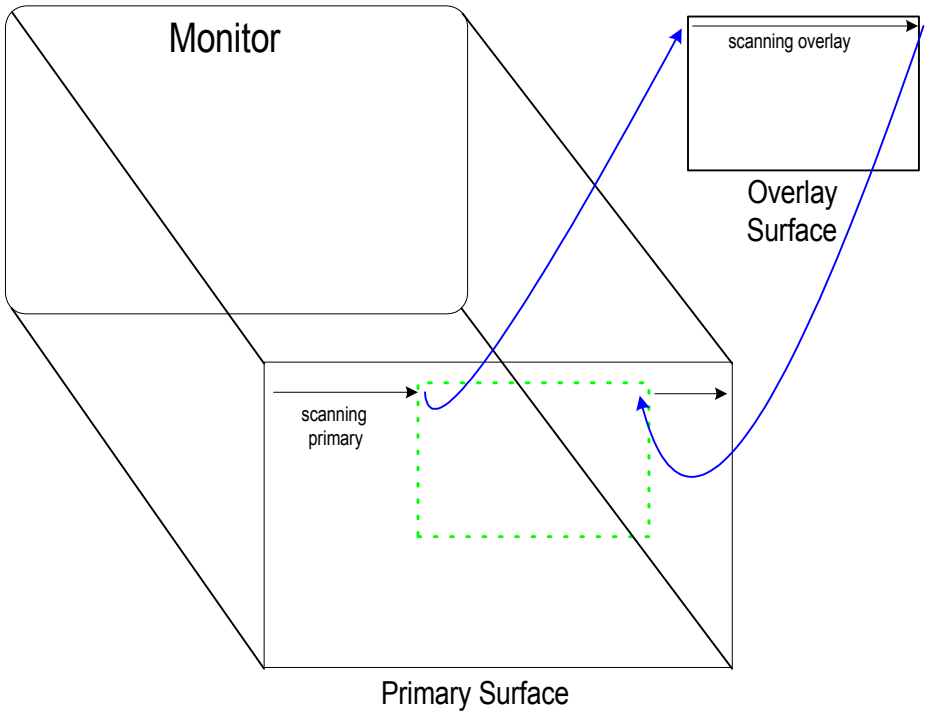
For information about implementing overlay surfaces, see Tutorial 6: Using Overlay Surfaces.

Overlay Surface Overview

Overlay surfaces, casually referred to as overlays, are surfaces with special hardware supported capabilities. Overlay surfaces are frequently used to display live video, recorded video, or still bitmaps over the primary surface without blitting to the primary surface or changing the primary surface's contents in any way. Overlay surface support is provided entirely by the hardware; DirectDraw supports any capabilities as reported by the display device driver. DirectDraw does not emulate overlay surfaces.

An overlay surface is analogous to a clear piece of plastic that you draw on and place in front of the monitor. When the overlay is in front of the monitor, you can see both the overlay and the contents of the primary surface together, but when you remove it, the primary surface's contents are unchanged. In fact, the mechanics of overlays work much like the clear plastic analogy. When you display an overlay surface, you're telling the device driver where and how you want it to be visible. While the display device paints scan lines to the monitor, it checks the location of each pixel in the

primary surface to see if an overlay should be visible there instead. If so, the display device substitutes data from the overlay surface for the corresponding pixel, as shown in the following illustration:



By using this method, the display adapter produces a composite of the primary surface and the overlay on the monitor, providing transparency and stretching effects, without modifying the contents of either surface. The composited surfaces are injected into the video stream and sent directly to the monitor. Because this on-the-fly processing and pixel substitution is handled at the hardware level, no noticeable performance loss occurs when displaying overlays. Additionally, this method makes it possible to seamlessly composite primary and overlay surfaces with different pixel formats.

You create overlay surfaces by calling the **IDirectDraw2::CreateSurface** method, specifying the DDSCAPS_OVERLAY flag in the associated **DDSCAPS** structure. Overlay surfaces can only be created in video memory, so you must also include the DDSCAPS_VIDEOMEMORY flag. As with other types of surfaces, by including the appropriate flags you can create either a single overlay or a flipping chain made up of multiple overlay surfaces.

Significant DDCAPS Members and Flags

You can retrieve information about the supported overlay features by calling the **IDirectDraw2::GetCaps** method. The method fills a **DDCAPS** structure with information describing all features.

When reporting hardware features, the device driver sets flags in the **dwCaps** structure member to indicate when a given type of restriction is enforced by the hardware. After retrieving the driver capabilities, examine the flags in the **dwCaps** member for information about which restrictions apply. The **DDCAPS** structure contains nine members that carry information describing hardware restrictions for overlay surfaces. The following table lists the overlay related members and their corresponding flags:

Member	Flag
dwMaxVisibleOverlays	This member is always valid
dwCurrVisibleOverlays	This member is always valid
dwAlignBoundarySrc	DDCAPS_ALIGNBOUNDARYSRC
dwAlignSizeSrc	DDCAPS_ALIGNSIZESRC
dwAlignBoundaryDest	DDCAPS_ALIGNBOUNDARYDEST
dwAlignSizeDest	DDCAPS_ALIGNSIZEDEST
dwMinOverlayStretch	DDCAPS_OVERLAYSTRETCH
dwMaxOverlayStretch	DDCAPS_OVERLAYSTRETCH

The **dwMaxVisibleOverlays** and **dwCurrVisibleOverlays** members carry information about the maximum number of overlays the hardware can display, and how many of them are currently visible.

Additionally, the hardware reports rectangle position and size alignment restrictions in the **dwAlignBoundarySrc**, **dwAlignSizeSrc**, **dwAlignBoundaryDest**, **dwAlignSizeDest**, and **dwAlignStrideAlign** members. The values in these members dictate how you must size and position source and destination rectangles when displaying overlay surfaces. For more information, see Source and Destination Rectangles and Boundary and Size Alignment.

Also, the hardware reports information about stretch factors in the **dwMinOverlayStretch** and **dwMaxOverlayStretch** members. For more information, see Minimum and Maximum Stretch Factors.

Source and Destination Rectangles

To display an overlay surface, you call the overlay surface's **IDirectDrawSurface3::UpdateOverlay** method, specifying the DDOVER_SHOW flag in the *dwFlags* parameter. The method requires you to specify a source and destination rectangle in the *lpSrcRect* and *lpDestRect* parameters. The source rectangle describes a rectangle on the overlay surface that will be visible on the primary surface. To request that the method use the entire surface, set the *lpSrcRect*

parameter to NULL. The destination rectangle describes a portion of the primary surface on which the overlay surface will be displayed.

Source and destination rectangles do not need to be the same size. You can often specify a destination rectangle smaller or larger than the source rectangle, and the hardware will shrink or stretch the overlay appropriately when it is displayed.

To successfully display an overlay surface, you might need to adjust the size and position of both rectangles. Whether this is necessary depends on the restrictions imposed by the device driver. For more information, see Boundary and Size Alignment and Minimum and Maximum Stretch Factors.

Boundary and Size Alignment

Due to various hardware limitations, some device drivers impose restrictions on the position and size of the source and destination rectangles used to display overlay surfaces. To find out which restrictions apply for a device, call the **IDirectDraw2::GetCaps** method and then examine the overlay-related flags in the **dwCaps** member of the **DDCAPS** structure. The following table shows the members and flags specific to boundary and size alignment restrictions:

Category	Flag	Member
Boundary (position) restrictions	DDCAPS_ALIGNBOUNDARYSRC	**dwAlignBoundary**
	DDCAPS_ALIGNBOUNDARYDEST	**dwAlignBoundary**
Size restrictions	DDCAPS_ALIGNSIZESRC	**dwAlignSizeSrc**
	DDCAPS_ALIGNSIZEDEST	**dwAlignSizeDest**

There are two types of restrictions, boundary restrictions and size restrictions. Both types of restrictions are expressed in terms of pixels (not bytes) and can apply to the source and destination rectangles. Also, these restrictions can vary depending on the pixel formats of the overlay and primary surface.

Boundary restrictions affect where you can position a source or destination rectangle. The values in the **dwAlignBoundarySrc** and **dwAlignBoundaryDest** members tell you how to align the top left corner of the corresponding rectangle. The x-coordinate of the top left corner of the rectangle (the **left** member of the **RECT** structure), must be a multiple of the reported value.

Size restrictions affect the valid widths for source and destination rectangles. The values in the **dwAlignSizeSrc** and **dwAlignSizeDest** members tell you how to align the width, in pixels, of the corresponding rectangle. Your rectangles must have a pixel width that is a multiple of the reported value. If you stretch the rectangle to comply with a minimum required stretch factor, be sure that the stretched rectangle is still size aligned. After stretching the rectangle, align its width by rounding up, not down, so you preserve the minimum stretch factor. For more information, see Minimum and Maximum Stretch Factors.

Minimum and Maximum Stretch Factors

Due to hardware limitations, some devices restrict how wide a destination rectangle can be compared with the corresponding source rectangle. DirectDraw communicates these restrictions as stretch factors. A stretch factor is the ratio between the widths of the source and destination rectangles. If the driver provides information about stretch factors, it sets the DDCAPS_OVERLAYSTRETCH flag in the **DDCAPS** structure after you call the **IDirectDraw2::GetCaps** method. Note that stretch factors are reported multiplied by 1000, so a value of 1300 actually means 1.3 (and 750 would be 0.75).

Devices that do not impose limits on stretching or shrinking an overlay destination rectangle often report a minimum and maximum stretch factor of 0.

The minimum stretch factor tells you how much wider or narrower than the source rectangle the destination rectangle needs to be. If the minimum stretch factor is greater than 1000, then you must increase the destination rectangle's width by that ratio. For instance, if the driver reports 1300, you must make sure that the destination rectangle's width is at least 1.3 times the width of the source rectangle. Similarly, a minimum stretch factor less than 1000 indicates that the destination rectangle can be smaller than the source rectangle by that ratio.

The maximum stretch factor tells the maximum amount you can stretch the width of the destination rectangle. For example, if the maximum stretch factor is 2000, you can specify destination rectangles that are up to, but not wider than, twice the width of the source rectangle. If the maximum stretch factor is less than 1000, then you must shrink the width of the destination rectangle by that ratio to be able to display the overlay.

After stretching, the destination rectangle must conform to any size alignment restrictions the device might require. Therefore, it's a good idea to stretch the destination rectangle before adjusting it to be size aligned. For more information, see Boundary and Size Alignment.

Hardware does not require that you adjust the height of destination rectangles. You can increase a destination rectangle's height to preserve aspect ratio without negative effects.

Overlay Color Keys

Like other types of surfaces, overlay surfaces use source and destination color keys for controlling transparent blit operations between surfaces. Because overlay surfaces are not displayed by blitting, there needs to be a different way to control how an overlay surface is displayed over the primary surface when you call the **IDirectDrawSurface3::UpdateOverlay** method. This need is filled by overlay color keys. Overlay color keys, like their blit-related counterparts, have a source version and a destination version that you set by calling the **IDirectDrawSurface3::SetColorKey** method. You use the DDCKEY_SRCOVERLAY or DDCKEY_DESTOVERLAY flags to set a source or destination overlay color key. Overlay surfaces can employ blit and overlay color

keys together to control blit operations and overlay display operations appropriately; the two types of color keys do not conflict with one another.

The **IDirectDrawSurface3::UpdateOverlay** method uses the source overlay color key to determine which pixels in the overlay surface should be considered transparent, allowing the contents of the primary surface to show through. Likewise, the method uses the destination overlay color key to determine the parts of the primary surface that will be covered up by the overlay surface when it is displayed. The resulting visual effect is the same as that created by blit-related color keys. For more information, see Transparent Blitting and Color Keys and Color Keying.

Positioning Overlay Surfaces

After initially displaying an overlay by calling the **IDirectDrawSurface3::UpdateOverlay** method, you can update the destination rectangle's by calling the **IDirectDrawSurface3::SetOverlayPosition** method.

Make sure that the positions you specify comply with any boundary alignment restrictions enforced by the hardware. For more information, see Boundary and Size Alignment. Also remember that **IDirectDraw2::SetOverlayPosition** doesn't perform clipping for you; using coordinates that would potentially make the overlay run off the edge of the target surface will cause the method to fail, returning DDERR_INVALIDPOSITION.

Creating Overlay Surfaces

Like all surfaces, you create an overlay surface by calling the **IDirectDraw2::CreateSurface** method. To create an overlay, include the DDSCAPS_OVERLAY flag in the associated **DDSCAPS** structure.

Overlay support varies widely across display devices. As a result, you cannot be sure that a given pixel format will be supported by most drivers and must therefore be prepared to work with a variety of pixel formats. You can request information about the non-RGB formats that a driver supports by calling the **IDirectDraw2::GetFourCCCodes** method.

When you attempt to create an overlay surface, it is advantageous to try creating a surface with the most desirable pixel format, falling back on other pixel formats if a given pixel format isn't supported.

You can create overlay surface flipping chains. For more information, see Creating Complex Surfaces and Flipping Chains.

Overlay Z-Orders

Overlay surfaces are assumed to be on top of all other screen components, but when you display multiple overlay surfaces, you need some way to visually organize them. DirectDraw supports *overlay z-ordering* to manage the order in which overlays clip each other. Z-order values represent conceptual distances from the primary surface toward the viewer. They range from 0, which is just on top of the primary surface, to

4 billion, which is as close to the viewer as possible, and no two overlays can share the same z-order. You set z-order values by calling the **IDirectDrawSurface3::UpdateOverlayZOrder** method.

Destination color keys are affected only by the bits on the primary surface, not by overlays occluded by other overlays. Source color keys work on an overlay whether or not a z-order was specified for the overlay.

Overlays without a specified z-order are assumed to have a z-order of 0. Overlays that do not have a specified z-order behave in unpredictable ways when overlaying the same area on the primary surface.

A DirectDraw object does not track the z-orders of overlays displayed by other applications.

Flipping Overlay Surfaces

Like other types of surfaces, you can create overlay flipping chains. After creating a flipping chain of overlays, call the **IDirectDrawSurface3::Flip** method to flip between them. For more information, see Flipping Surfaces.

Software decoders displaying video with overlay surfaces can use the DDFLIP_ODD and DDFLIP_EVEN flags when calling the **Flip** method to use features that reduce motion artifacts. If the driver supports odd-even flipping, the DDCAPS2_CANFLIPODDEVEN flag will be set in the **DDCAPS** structure after retrieving driver capabilities. If DDCAPS2_CANFLIPODDEVEN is set, you can include the DDOVER_BOB flag when calling the **IDirectDrawSurface3::UpdateOverlay** method to inform the driver that you want it to use the “Bob” algorithm to minimize motion artifacts. Later, when you call **Flip** with the DDFLIP_ODD or DDFLIP_EVEN flag, the driver will automatically adjust the overlay source rectangle to compensate for jittering artifacts.

If the driver doesn’t set the DDCAPS2_CANFLIPODDEVEN flag when you retrieve hardware capabilities, **UpdateOverlay** will fail if you specify the DDOVER_BOB flag.

For more information about the Bob algorithm, see Solutions to Common Video Artifacts.

Blitting to Multiple Windows

You can use a DirectDraw object and a DirectDrawClipper object to blit to multiple windows created by an application running at the normal cooperative level. For more information, see Using a Clipper with Multiple Windows.

Creating multiple DirectDraw objects that blit to each others’ primary surface is not recommended.

Palettes

This section contains information about DirectDrawPalette objects. The following topics are discussed:

- What Are Palettes?
- Palette Types
- Setting Palettes on Nonprimary Surfaces
- Sharing Palettes
- Palette Animation

What are Palettes?

Palettized surfaces need palettes to be meaningfully displayed. A palettized surface, also known as a color-indexed surface, is simply a collection of numbers where each number represents a pixel. The value of the number is an index into a color table that tells DirectDraw what color to use when displaying that pixel. DirectDrawPalette objects, casually referred to as palettes, provide you with an easy way to manage a color table. Surfaces that use a 16-bit or greater pixel format do not use palettes.

A DirectDrawPalette object represents an indexed color table that has 2, 4, 16 or 256 entries to be used with a color indexed surface. Each entry in the palette is an RGB triplet that describes the color to be used when displaying pixels within the surface. The color table can contain 16- or 24-bit RGB triplets representing the colors to be used. For 16-color palettes, the table can also contain indexes to another 256-color palette. Palettes are supported for textures, off-screen surfaces, and overlay surfaces, none of which is required to have the same palette as the primary surface.

You can create a palette by calling the **IDirectDraw2::CreatePalette** method. This method retrieves a pointer to the palette object's **IDirectDrawPalette** interface. You can use the methods of this interface to manipulate palette entries, retrieve information about the object's capabilities, or initialize the object (if you used the **CoCreateInstance** COM function to create it).

You apply a palette to a surface by calling the surface's **IDirectDrawSurface3::SetPalette** method. A single palette can be applied to multiple surfaces.

DirectDrawPalette objects reserve entry 0 and entry 255 for 8-bit palettes, unless you specify the DDPCAPS_ALLOW256 flag to request that these entries be made available to you.

You can retrieve palette entries by using the **IDirectDrawPalette::GetEntries** method, and you can change entries by using the **IDirectDrawPalette::SetEntries** method.

The Ddutil.cpp source file included with this SDK contains some handy application-defined functions for working with palettes. For more information, see the **DDLoadPalette** functions in that source file.

Palette Types

DirectDraw supports 1-bit (2 entry), 2-bit (4 entry), 4-bit (16 entry), and 8-bit (256 entry) palettes. A palette can only be attached to a surface that has a matching pixel format. For example, a 2-entry palette created with the DDPCAPS_1BIT flag can be attached only to a 1-bit surface created with the DDPF_PALETTEINDEXED1 flag.

Additionally, you can create palettes that don't contain a color table at all, known as index palettes. Instead of a color table, an index palette contains index values that represent locations another palette's color table.

To create an indexed palette, specify the DDPCAPS_8BITENTRIES flag when calling the **IDirectDraw2::CreatePalette** method. For example, to create a 4-bit indexed palette, specify both the DDPCAPS_4BIT and DDPCAPS_8BITENTRIES flags. When you create an indexed palette, you pass a pointer to an array of bytes rather than a pointer to an array of **PALETTEENTRY** structures. You must cast the pointer to the array of bytes to an **LPPALETTEENTRY** type when you use the **IDirectDraw2::CreatePalette** method.

Note that DirectDraw does not dereference index palette entries during blit operations.

Setting Palettes on Nonprimary Surfaces

Palettes can be attached to any palettized surface (primary, back buffer, off-screen plain, or texture map). Only those palettes attached to primary surfaces will have any effect on the system palette. It is important to note that DirectDraw blits never perform color conversion; any palettes attached to the source or destination surface of a blit are ignored.

Nonprimary surface palettes are intended for use by Direct3D applications.

Sharing Palettes

Palettes can be shared among multiple surfaces. The same palette can be set on the front buffer and the back buffer of a flipping chain or shared among multiple texture surfaces. When an application attaches a palette to a surface by using the **IDirectDrawSurface3::SetPalette** method, the surface increments the reference count of that palette. When the reference count of the surface reaches 0, the surface will decrement the reference count of the attached palette. In addition, if a palette is detached from a surface by using **IDirectDrawSurface3::SetPalette** with a NULL palette interface pointer, the reference count of the surface's palette will be decremented.

Note

If **IDirectDrawSurface3::SetPalette** is called several times consecutively on the same surface with the same palette, the reference count for the palette is incremented only once. Subsequent calls do not affect the palette's reference count.

Palette Animation

Palette animation refers to the process of modifying a surface's palette to change how the surface itself looks when displayed. By repeatedly changing the palette, the surface appears to change without actually modifying the contents of the surface. To this end, palette animation gives you a way to modify the appearance of a surface without changing its contents and with very little overhead.

There are two methods for providing straightforward palette animation:

- Modifying palette entries within a single palette
- Switching between multiple palettes

Using the first method, you change individual palette entries that correspond to the colors you want to animate, then reset the entries with a single call to the **IDirectDrawPalette::SetEntries** method.

The second method requires two or more DirectDrawPalette objects. When using this method, you perform the animation by attaching one palette object after another to the surface object by calling the **IDirectDrawSurface3::SetPalette** method.

Neither method is hardware intensive, so feel free to use whichever technique you see fit for your application.

For specific information and an example of how to implement palette animation, see Tutorial 5: Dynamically Modifying Palettes.

Clippers

This section contains information about DirectDrawClipper objects. The following topics are discussed:

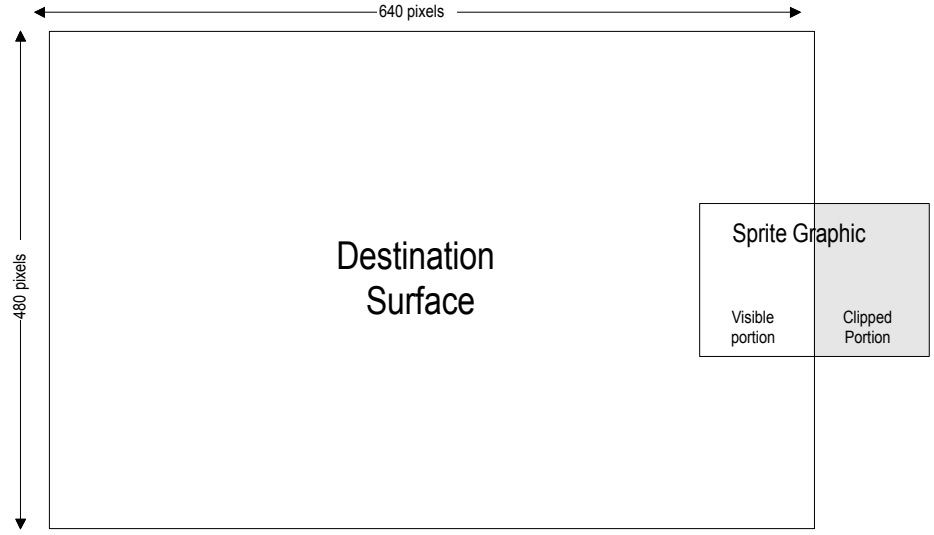
- What Are Clipper Objects?
- Clip Lists
- Sharing DirectDrawClipper Objects
- Independent DirectDrawClipper Objects
- Creating DirectDrawClipper Objects with CoCreateInstance
- Using a Clipper with the System Cursor
- Using a Clipper with Multiple Windows

What Are Clipper Objects?

Clippers, or DirectDrawClipper objects, allow you to blit to selected parts of a surface. A clipper object holds one or more clip lists. A clip list is one bounding rectangle or a list of several bounding rectangles that describe an area or areas of a surface to which you are allowed to blit. These areas are described with **RECT** structures, in screen coordinates.

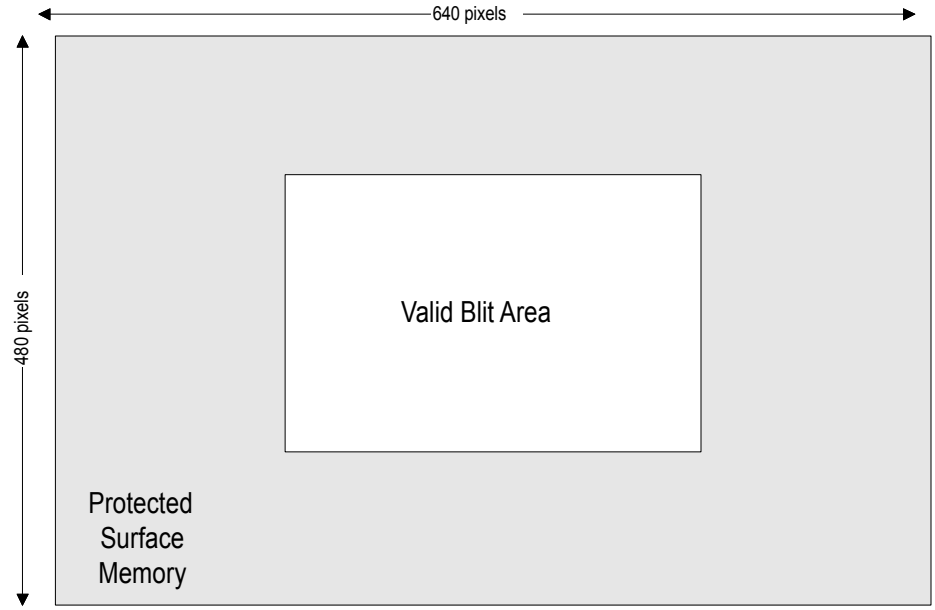
Clip lists are a very valuable tool. One common use for them is in preventing your application from blitting beyond the edges of the screen. For example, imagine that you want to display a sprite as it enters the screen from an edge. You don't want to make the sprite "pop" onto the screen; you want it to appear as though it is smoothly moving into view. Without a clipper object, you would need to include logic that restricts blit operations to protect surface memory that is logically off the edge of the screen. Failing to do this results in memory access violations.

The following illustration shows this type of clipping.



You can use clipper objects to designate certain areas within a destination surface as writable. DirectDraw clips blit operations in these areas, protecting the pixels outside the specified clipping rectangle.

The following illustration shows this clipping style.



Clip Lists

DirectDraw manages clip lists by using the DirectDrawClipper object. A clip list is a series of rectangles that describes the visible areas of the surface. A DirectDrawClipper object can be attached to any surface. A window handle can also be attached to a DirectDrawClipper object, in which case DirectDraw updates the DirectDrawClipper clip list with the clip list from the window as it changes.

Although the clip list is visible from the DirectDraw HAL, DirectDraw calls the HAL only for blitting with rectangles that meet the clip list requirements. For instance, if the upper-right rectangle of a surface was clipped and the application directed DirectDraw to blit the surface onto the primary surface, DirectDraw would have the HAL do two blits, the first being the upper-left corner of the surface, and the second being the bottom half of the surface.

Through the **IDirectDrawClipper::SetClipList** method, you can passing an entire clip list to the driver (if the driver supports this) rather than calling the driver multiple times, once for each rectangle in the clip list. Additionally, you can set the clipper to a single window by calling the **IDirectDrawClipper::SetHWnd** method, specifying the target window's handle. If you set a clipper using a window handle, you cannot set additional rectangles.

Clipping for overlay surfaces is supported only if the overlay hardware can support clipping and if destination color keying is not active.

Sharing DirectDrawClipper Objects

DirectDrawClipper objects can be shared between multiple surfaces. For example, the same DirectDrawClipper object can be set on both the front buffer and the back buffer of a flipping chain. When an application attaches a DirectDrawClipper object to a surface by using the **IDirectDrawSurface3::SetClipper** method, the surface increments the reference count of that object. When the reference count of the surface reaches 0, the surface will decrement the reference count of the attached DirectDrawClipper object. In addition, if a DirectDrawClipper object is detached from a surface by calling **IDirectDrawSurface3::SetClipper** with a NULL clipper interface pointer, the reference count of the surface's DirectDrawClipper object will be decremented.

Note

If **IDirectDrawSurface3::SetClipper** is called several times consecutively on the same surface for the same DirectDrawClipper object, the reference count for the object is incremented only once. Subsequent calls do not affect the object's reference count.

Independent DirectDrawClipper Objects

You can create DirectDrawClipper objects that are not directly owned by any particular DirectDraw object. These DirectDrawClipper objects can be shared across multiple DirectDraw objects. Driver-independent DirectDrawClipper objects are created by using the new **DirectDrawCreateClipper** DirectDraw function. An application can call this function before any DirectDraw objects are created.

Because DirectDraw objects do not own these DirectDrawClipper objects, they are not automatically released when your application's objects are released. If the application does not explicitly release these DirectDrawClipper objects, DirectDraw will release them when the application closes.

You can still create DirectDrawClipper objects by using the **IDirectDraw2::CreateClipper** method. These DirectDrawClipper objects are automatically released when the DirectDraw object from which they were created is released.

Creating DirectDrawClipper Objects with CoCreateInstance

DirectDrawClipper objects have full class-factory support for COM compliance. In addition to using the standard **DirectDrawCreateClipper** function and **IDirectDraw2::CreateClipper** method, you can also create a DirectDrawClipper object either by using the **CoGetClassObject** function to obtain a class factory and then calling the **CoCreateInstance** function, or by calling **CoCreateInstance** directly. The following example shows how to create a DirectDrawClipper object by using **CoCreateInstance** and the **IDirectDrawClipper::Initialize** method.

```
ddrval = CoCreateInstance(&CLSID_DirectDrawClipper,
   NULL, CLSCTX_ALL, &IID_IDirectDrawClipper, &lpClipper);
if (!FAILED(ddrval))
   ddrval = IDirectDrawClipper_Initialize(lpClipper,
      lpDD, 0UL);
```

In this call to **CoCreateInstance**, the first parameter, *CLSID_DirectDrawClipper*, is the class identifier of the DirectDrawClipper object class, the *IID_IDirectDrawClipper* parameter identifies the currently supported interface, and the *lpClipper* parameter points to the DirectDrawClipper object that is retrieved.

An application must use the **IDirectDrawClipper::Initialize** method to initialize DirectDrawClipper objects that were created by the class-factory mechanism before it can use the object. The value 0UL is the *dwFlags* parameter, which in this case has a value of 0 because no flags are currently supported. In the example shown here, *lpDD* is the DirectDraw object that owns the DirectDrawClipper object. However, you could supply a NULL value instead, which would create an independent DirectDrawClipper object. (This is equivalent to creating a DirectDrawClipper object by using the **DirectDrawCreateClipper** function.)

Before you close the application, shut down COM by using the **CoUninitialize** function.

Using a Clipper with the System Cursor

DirectDraw applications often need to provide a way for users to navigate using the mouse. For full screen exclusive mode applications that use page-flipping, the only option is to implement a mouse cursor manually with a sprite, moving the sprite based on data retrieved from the device by DirectInput® or by responding to Windows mouse messages. However, any application that doesn't use page-flipping can still use the system's mouse cursor support.

When you use the system mouse cursor, you will sometimes fall victim to graphic artifacts that occur when you blit to parts of the primary surface. These artifacts appear as portions of the mouse cursor seemingly left behind by the system.

A DirectDrawClipper object can prevent these artifacts from appearing by preventing the mouse cursor image from "being in the way" during a blit operation. It's a relatively simple matter to implement, as well. To do so, create a DirectDrawClipper object by calling the **IDirectDraw2::CreateClipper** method. Then, assign your application's window handle to the clipper with the **IDirectDrawClipper::SetHWnd** method. Once a clipper is attached, any subsequent blits you perform on the primary surface with the **IDirectDrawSurface3::Blt** method will not exhibit the artifact.

Note that the **IDirectDrawSurface3::BltFast** method, and its counterparts in the **IDirectDrawSurface** and **IDirectDrawSurface3** interfaces, will not work on surfaces with attached clippers.

Using a Clipper with Multiple Windows

You can use a DirectDrawClipper object to blit to multiple windows created by an application running at the normal cooperative level.

To do this, create a single DirectDraw object with a primary surface. Then, create a DirectDrawClipper object and assign it to your primary surface by calling the **IDirectDrawSurface3::SetClipper** method. To blit only the client area of a window, set the clipper to that window's client area by calling the **IDirectDrawClipper::SetHWnd** method before blitting to the primary surface. Whenever you need to blit to another window's client area, call the **IDirectDrawClipper::SetHWnd** method again with the new target window handle.

Creating multiple DirectDraw objects that blit to each others' primary surface is not recommended. The technique described above provides an efficient and reliable way to blit to multiple client areas with a single DirectDraw object.

Advanced DirectDraw Topics

This section supplements the DirectDraw overview, providing information about advanced DirectDraw issues. The following topics are discussed:

- Mode 13 Support
- Taking Advantage of DMA Support
- Using DirectDraw Palettes in Windowed Mode
- Working with Multiple Monitors
- Video-Ports
- Getting the Flip and Blit Status
- Blitting with Color Fill
- Determining the Capabilities of the Display Hardware
- Storing Bitmaps in Display Memory
- Triple Buffering
- DirectDraw Applications and Window Styles
- Matching True RGB Colors to the Frame Buffer's Color Space

Mode 13 Support

This section contains information about DirectDraw mode 13 graphics mode support. The following topics are discussed:

- About Mode 13
- Setting Mode 13
- Mode 13 and Surface Capabilities

- Using Mode 13

About Mode 13

DirectDraw supports access to the linear unflippable 320x200 8 bits per pixel palettized mode known widely by the name Mode 13, its hexadecimal BIOS mode number. DirectDraw treats this mode like a Mode X mode, but with some important differences imposed by the physical nature of Mode 13.

Setting Mode 13

Mode 13 has similar enumeration and mode-setting behavior as Mode X. DirectDraw will only enumerate Mode 13 if the DDSCL_ALLOWMODEX flag was passed to the **IDirectDraw2::SetCooperativeLevel** method.

You enumerate the Mode 13 display mode like all other modes, but you make a surface capabilities check before calling **IDirectDraw2::EnumDisplayModes**. To do this, call **IDirectDraw2::GetCaps** and check for the DDSCAPS_STANDARDVGAMODE flag in the **DDSCAPS** structure after the method returns. If this flag is not present, then Mode 13 is not supported, and attempts to enumerate with the DDEDM_STANDARDVGAMODES flag will fail, returning **DDERR_INVALIDPARAMS**.

The **EnumDisplayModes** method now supports a new enumeration flag, DDEDM_STANDARDVGAMODES, which causes DirectDraw to enumerate Mode 13 in addition to the 320x200x8 Mode X mode. There is also a new **IDirectDraw2::SetDisplayMode** flag, DDSDM_STANDARDVGAMODE, which you must pass in order to distinguish Mode 13 from 320x200x8 Mode X.

Note that some video cards offer linear accelerated 320x200x8 modes. On such cards DirectDraw will not enumerate Mode 13, enumerating the linear mode instead. In this case, if you attempt to set Mode 13 by passing the DDSDM_STANDARDVGAMODE flag to **SetDisplayMode**, the method will succeed, but the linear mode will be used. This is analogous to the way that linear low resolution modes override Mode X modes.

Mode 13 and Surface Capabilities

When DirectDraw calls an application's **EnumModesCallback** callback function, the **ddsCaps** member of the associated **DDSURFACEDESC** structure contains flags that reflect the mode being enumerated. You can expect DDSCAPS_MODEX for a Mode X mode or DDSCAPS_STANDARDVGAMODE for Mode 13. These flags are mutually exclusive. If neither of these bits is set, then the mode is a linear accelerated mode. This behavior also applies to the flags retrieved by the **IDirectDraw2::GetDisplayMode** method.

Using Mode 13

Because Mode 13 is a linear mode, DirectDraw can give an application direct access to the frame buffer. Unlike Mode X modes, you can call the **IDirectDrawSurface3::Lock**, **IDirectDrawSurface3::Blt**, and **IDirectDrawSurface3::BltFast** methods directly to access the primary surface.

When using Mode 13, DirectDraw supports an emulated **IDirectDrawSurface3::Flip** that is implemented as a straight copy of the contents of a back buffer to the primary surface. You can emulate this yourself by copying a smaller subrectangle of the back buffer to the primary using **Blt** or **BltFast**.

There is one caveat concerning Lock and Mode 13. Although DirectDraw allows direct linear access to the Mode 13 VGA frame buffer, do not assume that the buffer is always located at address 0xA0000, since DirectDraw can return an aliased virtual-memory pointer to the frame buffer which will not be 0xA0000. Similarly, do not assume that the pitch of a Mode 13 surface is 320, because display cards that support an accelerated 320x200x8 mode will very likely use a different pitch.

Taking Advantage of DMA Support

This section contains information about how you can take advantage of device support for Direct Memory Access (DMA) to increase performance in completing certain tasks. The following topics are discussed:

- About DMA Device Support
- Testing For DMA Support
- Typical Scenarios for DMA
- Using DMA

About DMA Device Support

Some display devices can perform blit operations (or other operations) on system memory surfaces. These operations are commonly referred to as Direct Memory Access (DMA) operations. You can exploit DMA support to accelerate certain combinations of operations. For example, on such a device, you could perform a blit from system memory to video memory while using the processor to prepare the next frame. In order to use such facilities, you must assume certain responsibilities. This section details these tasks.

Testing For DMA Support

Before using DMA operations, you must test the device for DMA support and, if it does support DMA, how much support it provides. Begin by retrieving the driver capabilities by calling the **IDirectDraw2::GetCaps** method, then look for the DDCAPS_CANBLTSYSMEM flag in the **dwCaps** member of the associated **DDCAPS** structure. If the flag is set, the device supports DMA.

If you know that DMA is generally supported, you also need to find out how well the driver supports it. You do so by looking at some other structure members that provide information about system-to-video, video-to-system, and system-to-system blit operations. These capabilities are provided in 12 **DDCAPS** structure members that are named according to blit and capability type. The following table shows these new members.

System-to-video	Video-to-system	System-to-system
dwSVBCaps	**dwVSBCaps**	**dwSSBCaps**
dwSVBCKeyCaps	**dwVSBCKeyCaps**	**dwSSBCKeyCaps**
dwSVBFXCaps	**dwVSBFXCaps**	**dwSSBFXCaps**
dwSVBRops	**dwVSBRops**	**dwSSBRops**

For example, the system-to-video blit capability flags are provided in the **dwSVBCaps**, **dwSVBCKeyCaps**, **dwSVBFXCaps** and **dwSVBRops** members. Similarly, video-to-system blit capabilities are in the members whose names begin with "**dwVSB**," and system-to system capabilities are in the "**dwSSB**" members. Examine the flags present in these members to determine the level of hardware support for that blit category.

The flags in these members are parallel with the blit-related flags included in the **dwCaps**, **dwCKeyCaps**, and **dwFXCaps** members, with respect to that member's blit type. For example, the **dwSVBCaps** member contains general blit capabilities as specified by the same flags you might find in the **dwCaps** member. Likewise, the raster operation values in the **dwSVBRops**, **dwVSBRops**, and **dwSSBRops** members provide information about the raster operations supported for a given type of blit operation.

One of the key features to look for in these members is support for asynchronous DMA blit operations. If the driver supports asynchronous DMA blits between surfaces, the DDCAPS_BLTQUEUE flag will be set in the **dwSVBCaps**, **dwVSBCaps**, or **dwSSBCaps** member. (Generally, you'll see the best support for system-memory-to-video-memory surfaces.) If the flag isn't present, the driver isn't reporting support for asynchronous DMA blit operations.

Typical Scenarios For DMA

System memory to video memory SRCCOPY transfers are the most common type of hardware-supported blit operation. Consequently, the most typical use for such an operation is to move textures from a large collection of system memory surfaces to a surface in video memory in preparation for subsequent operations. System-to-video DMA transfers are about as fast as processor-controlled transfers (for example, HEL blits), but are of great utility since they can operate in parallel with the host processor.

Using DMA

Hardware transfers use physical memory addresses, not the virtual addresses which are home to applications. Some device drivers require that you provide the surface's

physical memory address. This mechanism is implemented by the **IDirectDrawSurface3::PageLock** method. If the device driver does not require page locking, the DDCAPS2_NOPAGELOCKREQUIRED flag will be set when you retrieve the hardware capabilities by calling the **IDirectDraw2::GetCaps** method.

Page locking a surface prevents the system from committing a surface's physical memory to other uses, and guarantees that the surface's physical address will remain constant until a corresponding **IDirectDrawSurface3::PageUnlock** call is made. If the device driver requires page locking, DirectDraw will only allow DMA operations on system memory surfaces that the application has page locked. If you do not call **IDirectDrawSurface3::PageLock** in such a situation, DirectDraw will perform the transfers by using software emulation. Note that locking a large amount of system memory will make Windows run poorly. Therefore, it is highly recommended that only full-screen exclusive mode applications use **IDirectDrawSurface3::PageLock** for large amounts of system memory, and that such applications take care to unlock these surfaces when the application is minimized. Of course, when the application is restored, you should page lock the system memory surface again.

Responsibility for managing page locking is entirely in the hands of the application developer. DirectDraw will never page lock or page unlock a surface. Additionally, it is up to you to determine how much memory you can safely page lock without adversely affecting system performance.

Using DirectDraw Palettes in Windowed Mode

IDirectDrawPalette interface methods write directly to the hardware when the display is in exclusive (full-screen) mode. However, when the display is in nonexclusive (windowed) mode, the **IDirectDrawPalette** interface methods call the GDIs palette handling functions to work cooperatively with other windowed applications.

The discussion in the following topics assumes that the desktop is in an 8-bit palettized mode and that you have created a primary surface and a typical window.

- Types of Palette Entries in Windowed Mode
- Creating a Palette in Windowed Mode
- Setting Palette Entries in Windowed Mode

Types of Palette Entries in Windowed Mode

Unlike full-screen exclusive mode applications, windowed applications must share the desktop palette with other applications. This imposes several restrictions on which palette entries you can safely modify and how you can modify them. The **PALETTEENTRY** structure you use when working with DirectDrawPalette objects and GDI contains a **peFlags** member to carry information that describes how the system should interpret the **PALETTEENTRY** structure.

The **peFlags** member describes three types of palette entries, discussed in this topic:

- Windows static entries
- Animated entries
- Nonanimated entries

Windows static entries.

In normal mode, Windows reserves palette entries 0 through 9 and 246 through 255 for system colors that it uses to display menu bars, menu text, window borders, and so on. In order to maintain a consistent look for your application and avoid damaging the appearance of other applications, you need to protect these entries in the palette you set to the primary surface. Often, developers retrieve the system palette entries by calling the **GetSystemPaletteEntries** Win32® function, then explicitly set the identical entries in a custom palette to match before assigning it to the primary surface. Duplicating the system palette entries in a custom palette will work initially, but it becomes invalid if the user changes the desktop color scheme.

To avoid having your palette look bad when the user changes color schemes, you can protect the appropriate entries by providing a reference into the system palette instead specifying a color value. This way, no matter what color the system is using for a given entry, your palette will always match and you won't need to do any updating. The PC_EXPLICIT flag, used in the **peFlags** member, makes it possible for you to directly refer to a system palette entry. When you use this flag, the system no longer assumes that the other structure members include color information. Rather, when you use PC_EXPLICIT, you set the value in the **peRed** member to the desired system palette index and set the other colors to zero.

For instance, if you want to ensure that the proper entries in your palette always match the system's color scheme, you could use the following code:

```
// Set the first and last 10 entries to match the system palette.
PALETTEENTRY pe[256];
ZeroMemory(pe, sizeof(pe));
for(int i=0;i<10;i++){
   pe[i].peFlags  = pe[i+246].peFlags = PC_EXPLICIT;
   pe[i].peRed = i;
   pe[i+246].peRed = i+246;
}
```

You can force Windows to use only the first and last palette entry (0 and 255) by calling the **SetSystemPaletteUse** Win32 function. In this case, you should set only entries 0 and 255 of your **PALETTEENTRY** structure to PC_EXPLICIT.

Animated entries

You specify palette entries that you will be animating by using the PC_RESERVED flag in the corresponding PALETTEENTRY structure. Windows will not allow any other application to map its logical palette entry to that physical entry, thereby preventing other applications from cycling their colors when your application animates the palette.

Nonanimated entries

You specify normal, nonanimated palette entries by using the PC_NOCOLLAPSE flag in the corresponding **PALETTEENTRY** structure. The PC_NOCOLLAPSE flag informs Windows not to substitute some other already-allocated physical palette entry for that entry.

Creating a Palette in Windowed Mode

The following example illustrates how to create a DirectDraw palette in nonexclusive (windowed) mode. In order for your palette to work correctly, it is vital that you set up every one of the 256 entries in the **PALETTEENTRY** structure that you submit to the **IDirectDraw2::CreatePalette** method.

```
LPDIRECTDRAW       lpDD; // Assumed to be initialized previously
PALETTEENTRY       pPaletteEntry[256];
int                index;
HRESULT            ddrval;
LPDIRECTDRAWPALETTE lpDDPal;

// First set up the Windows static entries.
for (index = 0; index < 10 ; index++)
{
   // The first 10 static entries:
   pPaletteEntry[index].peFlags = PC_EXPLICIT;
   pPaletteEntry[index].peRed = index;
   pPaletteEntry[index].peGreen = 0;
   pPaletteEntry[index].peBlue = 0;

   // The last 10 static entries:
   pPaletteEntry[index+246].peFlags = PC_EXPLICIT;
   pPaletteEntry[index+246].peRed = index+246;
   pPaletteEntry[index+246].peGreen = 0;
   pPaletteEntry[index+246].peBlue = 0;
}

// Now set up private entries. In this example, the first 16
// available entries are animated.
for (index = 10; index < 26; index ++)
{
   pPaletteEntry[index].peFlags = PC_NOCOLLAPSE|PC_RESERVED;
   pPaletteEntry[index].peRed = 255;
   pPaletteEntry[index].peGreen = 64;
   pPaletteEntry[index].peBlue = 32;
}

// Now set up the rest, the nonanimated entries.
```

```
for (; index < 246; index ++) // Index is set up by previous for loop
{
   pPaletteEntry[index].peFlags = PC_NOCOLLAPSE;
   pPaletteEntry[index].peRed = 25;
   pPaletteEntry[index].peGreen = 6;
   pPaletteEntry[index].peBlue = 63;
}

// All 256 entries are filled. Create the palette.
ddrval = lpDD->CreatePalette(DDPCAPS_8BIT, pPaletteEntry,
   &lpDDPal,NULL);
```

Setting Palette Entries in Windowed Mode

The rules that apply to the **PALETTEENTRY** structure used with the **IDirectDraw2::CreatePalette** method also apply to the **IDirectDrawPalette::SetEntries** method. Typically, you maintain your own array of **PALETTEENTRY** structures, so you do not need to rebuild it. When necessary, you can modify the array, and then call **IDirectDrawPalette::SetEntries** when it is time to update the palette.

In most circumstances, you should not attempt to set any of the Windows static entries when in nonexclusive (windowed) mode or you will get unpredictable results. The only exception is when you reset the 256 entries.

For palette animation, you typically change only a small subset of entries in your **PALETTEENTRY** array. You submit only those entries to **IDirectDrawPalette::SetEntries**. If you are resetting such a small subset, you must reset only those entries marked with the PC_NOCOLLAPSE and PC_RESERVED flags. Attempting to animate other entries can have unpredictable results.

The following example illustrates palette animation in nonexclusive mode:

```
LPDIRECTDRAW       lpDD;       // Already initialized
PALETTEENTRY pPaletteEntry[256]; // Already initialized
LPDIRECTDRAWPALETTE lpDDPal;     // Already initialized
int                index;
HRESULT            ddrval;
PALETTEENTRY       temp;

// Animate some entries. Cycle the first 16 available entries.
// They were already animated.
temp = pPaletteEntry[10];
for (index = 10; index < 25; index ++)
{
   pPaletteEntry[index] = pPaletteEntry[index+1];
}
pPaletteEntry[25] = temp;
```

```
// Set the values. Do not pass a pointer to the entire palette entry
// structure, but only to the changed entries.
ddrval = lpDDPal->SetEntries(
    0,                  // Flags must be zero
    10,                  // First entry
    16,                  // Number of entries
    & (pPaletteEntry[10])); // Where to get the data
```

Working with Multiple Monitors

Future releases of Windows 95, code named Memphis, and Windows NT support multiple display devices and monitors on a single system. The multiple monitor architecture (casually referred to as "MultiMon") enables the operating system to use the display area from two or more display devices and monitors to create a single logical desktop. For example, in a MultiMon system with two monitors, the user could display applications on either monitor, or even drag windows from one monitor to another. DirectDraw supports this architecture, but there are a few nuances to be aware of, depending on the cooperative level your application uses.

A DirectDraw application should enumerate the devices, choose a device (or perhaps allow the user to choose the device to use), then create a DirectDraw object for that the device by using its hardware globally unique identifier (GUID). This technique will ensure the best performance on both MultiMon and single monitor systems and at all cooperative levels.

The currently active display device is referred to as the "default device," or the "null device." The latter name comes from the fact that the currently active display device is enumerated with NULL as its GUID. Many existing applications create a DirectDraw object for the null device, assuming that the device will be hardware accelerated. However, on multiple monitor systems, the null device isn't always hardware accelerated; it depends on what cooperative level is set at the time.

In full-screen exclusive mode, the null device is hardware accelerated, but unaware of any other installed devices. This means that full-screen, exclusive mode applications will run as fast on a MultiMon system as any other system, but will not be able to use built-in support for spanning graphics operations across display devices. Full-screen, exclusive mode applications that need to use multiple devices can create a DirectDraw object for each device they want to use. Note that to create a DirectDraw object for a specific device, you must supply that device's GUID (as it is enumerated when you call **DirectDrawEnumerate**).

When the normal cooperative level is set, the null device has no hardware acceleration; the null device is, effectively, an emulated logical device that combines the resources of two physical devices. Therefore, the null device has no hardware acceleration at all when the normal cooperative level is set. On the other hand, when the normal cooperative level is set, the null device is capable of automatically spanning graphics operations across monitors. As a result, negative coordinates for

blit operations are valid when the logical location of secondary monitor is to the left of the primary monitor.

If your application requires hardware acceleration when the normal cooperative level is set, it must create a single DirectDraw object using a specific device's GUID. Note that when you don't use the null device, you don't get automatic device spanning. That is, blit operations that cross an edge of the primary surface will be clipped (if you are using a clipper) or will fail, returning DDERR_INVALIDRECT.

As a rule on any system, you should set the cooperative level immediately after creating a DirectDraw object, before retrieving the object's capabilities or querying for other interfaces. Additionally, avoid setting the cooperative level multiple times on a MultiMon system. If you need to switch from full-screen to normal mode, it is best to create a new DirectDraw object.

Video Ports

DirectDraw video-port extensions are a low-level programming interface, not intended for mainstream multimedia programmers. The target customer is the video-streaming software industry, which creates products like DirectShow™. Developers who want to include video playback in their software can make use of video-port extensions. However, for most software, a high-level programming interface like the one provided by DirectShow is recommended for greater ease of use.

This section contains information about DirectDrawVideoPort objects. The following topics are discussed:

- What is a Video-Port Object?
- Video-Port Technology Overview
- About DirectDraw Video-Port Extensions
- Video Frames and Fields
- HREF, VREF, and Connections
- Vertical Blanking Interval Data
- Auto-Flipping
- Solutions to Common Video Artifacts
- Solving Problems Caused by Half-Lines
- Exploiting Hardware Features

What is a Video-Port Object?

A DirectDrawVideoPort object represents the video-port hardware found on some display adapters. Generally, a video-port object controls how the video-port hardware applies a video signal it receives from a video decoder directly to the frame buffer.

More than one channel of video can be controlled by creating as many DirectDrawVideoPort objects as is required. Because each channel can be separately

enumerated and configured, the video hardware for each channel does not need to be identical.

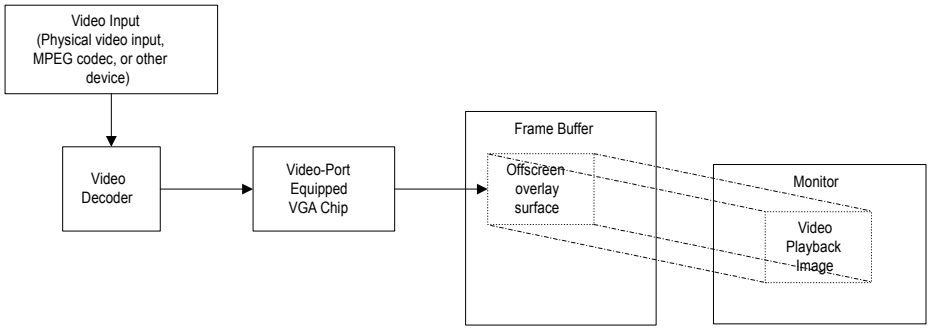
For more information, see Video-Port Technology Overview.

Video-Port Technology Overview

A video port is hardware on a display device that enables direct access to a surface within the frame buffer, bypassing the CPU and PCI bus. Direct frame buffer access makes it possible to efficiently play live or recorded video without creating noticeable load on the CPU. Once in a surface, an image can be displayed on the screen as an overlay, used as a Direct3D texture, or accessed by the CPU for capture or other processing. The following paragraphs provide general information about the components that make up the technology and how they work.

Data Flow

In a machine equipped with a video port, data in a video stream can flow directly from a video source through a video decoder and the video port to the frame buffer. These components often exist together on a display adapter, but can be on separate hardware components that are physically connected to one another. An example of this data flow is provided in the following illustration.



Video source

In the scope of video-port technology, a video source is strictly a hardware video input device, such as a Zoom Video port, MPEG codec, or other hardware source. These sources broadcast signals in a variety of formats, including NTSC, PAL, and SECAM through a physical connection to a video decoder.

Video Decoder

A video decoder is also a hardware component. The video decoder's job is to decipher the information provided by the video source and send it to the video port in an agreed upon connection format. The decoder possesses a physical connection to the video port, and exposes its services through a stream class minidriver. The decoder is responsible for sending video data and clock and sync information to the video port.

Video port

Like the other components in the data flow path, the video port is a piece of hardware. The video port exists on the display adapter's VGA chip and has direct access to the frame buffer. It receives information sent from the decoder, processes it, and places it in the frame buffer to be displayed. During processing, the video port can manipulate image data to provide scaling, shrinking, color control, or cropping services.

Frame Buffer

The frame buffer accepts video data as provided by the video port. Once received, applications can programmatically manipulate the image data, blit it to other locations, or show it on the display using an overlay (the most common function).

About DirectDraw Video-Port Extensions

DirectDraw has been extended to include the DirectDrawVideoPort object, which takes advantage of video-port technology and provides its services through the **IDDVideoPortContainer** and **IDirectDrawVideoPort** interfaces.

DirectDrawVideoPort objects do not control the video decoder, because it provides services of its own, nor does DirectDraw control the video source; it is beyond the scope of the video port. Rather, a DirectDrawVideoPort object represents the video port itself. It monitors the incoming signal and passes image data to the frame buffer, using parameters set though its interface methods to modify the image, perform flipping, or carry out other services.

The **IDDVideoPortContainer** interface, which you can retrieve by calling the **IDirectDraw2::QueryInterface** method, provides methods to query the hardware for its capabilities and create video-port objects. You create a video-port object by calling the **IDDVideoPortContainer::CreateVideoPort** method. Video-port objects expose their functionality through the **IDirectDrawVideoPort** interface, enabling you to manipulate the video-port hardware itself. Using these interfaces, you can examine the video-port's capabilities, assign an overlay surface to receive image data, start and stop video playback, and set hardware parameters to manipulate image data for cropping, color control, scaling, or shrinking effects.

DirectDraw video-port extensions provide for multiple video ports on the same machine by allowing you to create multiple DirectDrawVideoPort objects. There is no requirement that multiple video ports on a machine be identical—each port is separately enumerated and configured separately, regardless of any hardware differences that might exist.

In keeping with the general philosophy of DirectX, this technology gives programmers low-level access to hardware features while insulating them from specific hardware implementation details. It is not a high-level API.

Video Frames and Fields

Video can be interlaced or non-interlaced. When a video signal is interlaced, each video frame is made of two fields of image data. Each field is a collection of every other scan line in an image, starting with the first or second scan line. The first field,

referred to as the odd field (or field 1), contains the data for the first scan line and skips every other scan line to the end of the image. Similarly, the even field (or field 2), carries every other scan line starting with the second. The “even-ness” or “odd-ness” of a field is referred to as its field polarity.

When video is not interlaced, each field contains all of a frame's scan lines. Typically, video signals are sent at a rate of 30 frames per second; in the case of interleaved video, this means the rate is 60 fields per second.

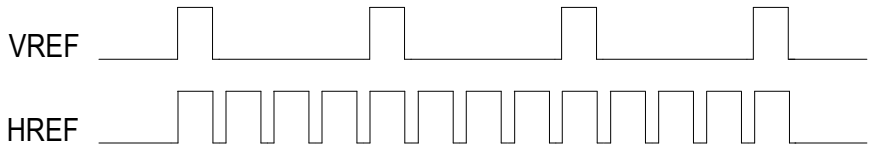
The fields that make up a frame do not always reflect the same moment in time. For example, if the frames are separated by 1/30 of a second then the two fields of a frame may be separated by 1/60 of a second. Because a television displays each field individually, no two fields are simultaneously visible, and the difference between fields adds to the illusion of movement.

HREF, VREF, and Connections

When a monitor or other display device is displaying an image, it typically scans down the screen, creating an image from left to right, top to bottom. (Sometimes, the device makes two passes down the screen to create a single image; this type of display is called an interlaced display.) The video stream contains signals that instruct the display device when a new line or new screen is to be drawn.

The terms HREF and VREF, also known as hsync and vsync, are the signals within the video stream that tell a display device what to do and when to do it. The HREF signals that a new line is to be drawn and the VREF signals a new screen.

For instance, imagine you’re working with a video signal intended for the world’s smallest monitor. The monitor only has 4 scan lines. (This is not at all realistic, of course, but it’s simple.) On an oscilloscope, the HREF and VREF signals would look somewhat like the following illustration:



In the preceding illustration, both HREF and VREF signals are “active high,” meaning that they are considered active when in a heightened state (when the waves go up). There is no standard for these signals. In some cases, places where the waves go down (“low” states) might signal an active HREF or VREF, or sometimes one will be active high and the other active low. Although the preceding illustration is only an imaginary example, note that there are lots of HREF signals for each VREF. This is because for each new screen, there are several scan lines. Of course, in a real video signal for a real broadcast, you would see hundreds of HREFs for a single VREF.

HREF signals, VREF signals, and video data are carried across physical data lines from the decoder to the video port. In many cases, a number of lines are reserved for

video data, and others are dedicated to carrying HREF and VREF signals. However, there is no standard for how these data lines are used.

A connection is a protocol that a video port or decoder uses to define how it uses these data lines. Video ports and video decoders will support a variety of connections. DirectDraw video-port extensions use globally-unique identifiers (GUIDs) to identify each type of connection. You can query for the connections that the video port supports by calling the **IDDVideoPortContainer::GetVideoPortConnectInfo** method. You create a DirectDrawVideoPort object that supports a given connection by calling the **IDDVideoPortContainer::CreateVideoPort** method.

Keep in mind that the video decoder is outside the scope of DirectDraw video-port extensions, and exposes its supported connections through an interface of its own. By enumerating the connections that the video-port supports and comparing the results with the connections supported by the decoder, you can negotiate a common connection (or "language") that both components understand.

Vertical Blanking Interval Data

In broadcast video, a small period of time elapses between video frames, during which a display device refreshes its display for the next frame. This period of time is called the Vertical Blanking Interval (VBI). Instead of sitting idle during the VBI, broadcast video encodes data in the first twenty-one scan lines of a video frame and sends these lines during the VBI. This data is often used for closed captioning or time-stamping, but can be used for other purposes.

DirectDraw video-port extensions enable you to divert data contained with the VBI to a surface, bypass scaling of VBI data, and automatically flip between VBI surfaces in a flipping chain. Once data is in a surface, you can directly access the surface's memory as needed.

For more information, see Auto-flipping.

Auto-flipping

To avoid tearing images when refreshing the screen between frames, DirectDrawVideoPort objects can automatically flip their target overlay surfaces in response to VREF signals. To use this service, the target surface you set to the video-port object with the **IDirectDrawVideoPort::SetTargetSurface** method must be the first surface in a flipping chain of overlay surfaces. Then, to begin playing the video sequence, call the **IDirectDrawVideoPort::StartVideo** method, specifying the DDVP_AUTOFLIP flag in the **dwVPFlags** member of the associated **DDVIDEOPORTINFO** structure. The video-port object will flip to the next surface in the flipping chain for each VREF signal is receives. If the video port is interleaving fields, it will flip once for every two VREF signals it receives.

If you are using auto-flipping and want to direct VBI data to separate auto-flipped surfaces, you must have the same number of VBI surfaces as you do standard video surfaces.

Solutions to Common Video Artifacts

Several problems are inherent in displaying broadcast video on display devices other than televisions. This section briefly discusses some common problems, then describes how DirectDraw video-port extensions tries to solve them.

NTSC Interlaced Display and Interleaved Memory

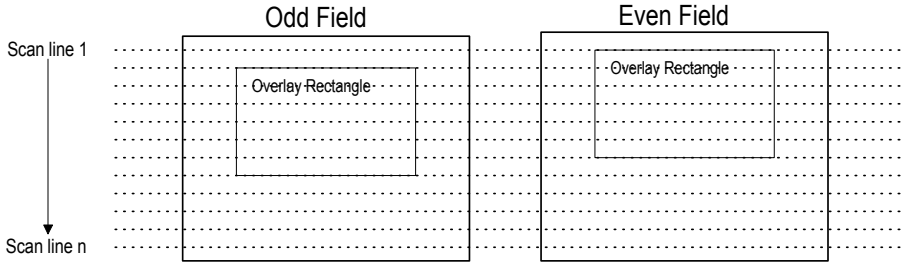
An NTSC signal broadcasts video at an approximate rate of 30 frames, or 60 fields, per second. Like a frame, a field in an NTSC signal is independent of the other field in a frame and can contain different image data. For more information on this behavior, see Video Frames and Fields.

The problems caused by the independence of fields within a frame become apparent when two fields are interleaved for display. In video with a lot of movement, the two fields of a single frame will contain images that don't match each other, resulting in motion artifacts.

One way that developers have tried to work around this behavior is by discarding one of the fields. This solution causes a loss in image quality by roughly one-half, but provides acceptable results for some purposes. Another method frequently used is to display fields individually, stretching each vertically by a factor of two when it is displayed. This provides better image quality, but because fields are offset by one pixel in the Y direction, the result is an animation that "jitters" up and down as it plays.

DirectDraw video-port extensions can employ two, more advanced, techniques for improving image quality, known as "Bob" and "Weave." Both are supported by the DirectDraw overlay surfaces that are used with video-port extensions.

The first algorithm, "Bob," is very similar to the method of displaying each field in a frame individually. However, for each field, the overlay's source rectangle is adjusted to accommodate for any jittering effects. Effectively, the source rectangle bounces up and down in time with the fields, negating the jittering onscreen. The following illustration depicts this process.



The "Weave" algorithm provides the best image quality for material that originates from film by exploiting a common technique used in the video industry for converting motion pictures to television. Unlike Bob, a video-port object does not Weave by itself; you must combine the default overlay behavior of displaying both fields

simultaneously with kernel mode video transport (to be provided a future release of Windows 95, code named Memphis, and Windows NT) to implement the algorithm.

Here is a synopsis of the algorithm, provided for completeness. Motion pictures capture video at a rate of 24 frames per second. When converting a motion picture for television, technicians use a technique called "3:2 pulldown" to convert the frame rate to the 30 frames per second required for television broadcasts. This technique involves inserting a redundant field for every four true fields in the video stream to come up with the required number of fields.

When you "weave," you are reversing this process. You detect when 3:2 pulldown is being used, removing any redundant fields to restore the original motion-picture frames. The fields that make up the restored frames can then be interleaved in memory without risk of motion artifacts. Occasionally, the pattern of redundant frames will change due to edits within the original film or reel breaks. You must monitor when these changes occur and update the behavior to adjust for the new pattern.

By default, an overlay surface displays both fields simultaneously. This works well if you're implementing the Weave algorithm, but prevents the video port from using the Bob algorithm. You can programmatically change how the overlay treats video data by calling the **IDirectDrawSurface3::UpdateOverlay** method. The flags you include in the **dwFlags** parameter determine the overlay's behavior: if you include the DDOVER_BOB flag, the video port will use the Bobbing algorithm; if you don't, it displays both fields. Note that by simply displaying both fields simultaneously, the resulting video will show motion artifacts.

Solving Problems Caused by Half-Lines

Some video decoders output a half line of meaningless data at the beginning of the even field. If this extra line is written to the frame buffer, the resulting image will appear garbled. In some cases, the video-port hardware is capable of sensing and discarding this data before writing it to the frame buffer.

You can determine if a video port is capable of discarding this data when retrieving connection information with the **IDDVideoPortContainer::GetVideoPortConnectInfo** method. If the video port cannot discard half-lines, the DDVPCONNECT_HALFLINE flag will be specified in the **dwFlags** member of the associated **DDVIDEOPORTCONNECT** structure for each supported connection.

If the video port is unable to discard half-lines, you have two options: you can discard one of the fields, or you can work around the hardware's limitations by making some adjustments in how you create the video-port object, and display images with the target overlay surface

Here's how to work around the problem. When creating the video-port object by calling the **IDDVideoPortContainer::CreateVideoPort** method, include the DDVPCONNECT_INVERTPOLARITY flag in the **dwFlags** member of the associated **DDVIDEOPORTCONNECT** structure. This causes the video port to

invert the polarity of the fields in the video stream, treating even fields like odd fields and vice versa. Once reversed, the half-line preceding even fields will be written to the frame buffer as the first scan line of each frame. To remove the unwanted data, adjust the source rectangle of the overlay surface used to display the image down one pixel by calling the **IDirectDrawVideoPort::StartVideo** method with the necessary coordinates. Note that this technique requires that you allocate one extra line in the surface containing the even field.

Exploiting Hardware Features

Video-port hardware often supports special features for adjusting color, shrinking or zooming images, handling VBI data, or skipping fields. The HAL provides information about these features by using flags in the **DDVIDEOPORTCAPS** structure. You retrieve the capabilities of a machine's video-port hardware by calling the **IDDVideoPortContainer::EnumVideoPorts** method.

To exploit these features for playback, you use the **IDirectDrawVideoPort::StartVideo** method, which uses a **DDVIDEOPORTINFO** structure to request that hardware features be used to modify image data before placing it in the frame buffer or for display. By setting values and flags in this structure, you can specify the source rectangle used with the overlay surface, indicate cropping regions, request hardware scaling, and set pixel formats.

DirectDrawVideoPort objects do not emulate video-port hardware services.

Getting the Flip and Blit Status

When the **IDirectDrawSurface3::Flip** method is called, the primary surface and back buffer are exchanged. However, the exchange may not occur immediately. For example, if a previous flip has not finished, or if it did not succeed, this method returns **DDERR_WASSTILLDRAWING**. In the samples included with this SDK, the **IDirectDrawSurface3::Flip** call continues to loop until it returns DD_OK. Also, a **IDirectDrawSurface3::Flip** call does not complete immediately. It schedules a flip for the next time a vertical blank occurs on the system.

An application that waits until the **DDERR_WASSTILLDRAWING** value is not returned is very inefficient. Instead, you could create a function in your application that calls the **IDirectDrawSurface3::GetFlipStatus** method on the back buffer to determine if the previous flip has finished.

If the previous flip has not finished and the call returns **DDERR_WASSTILLDRAWING**, your application can use the time to perform another task before it checks the status again. Otherwise, you can perform the next flip. The following example demonstrates this concept:

```
while(lpDDSBack->GetFlipStatus(DDGFS_ISFLIPDONE) ==
    DDERR_WASSTILLDRAWING);

    // Waiting for the previous flip to finish. The application can
```

```
    // perform another task here.

ddrval = lpDDSPrimary->Flip(NULL, 0);
```

You can use the **IDirectDrawSurface3::GetBltStatus** method in much the same way to determine whether a blit has finished. Because **IDirectDrawSurface3::GetFlipStatus** and **IDirectDrawSurface3::GetBltStatus** return immediately, you can use them periodically in your application with little loss in speed.

Blitting with Color Fill

You can use the **IDirectDrawSurface3::Blt** method to perform a color fill of the most common color you want to be displayed. For example, if the most common color your application displays is blue, you can use **IDirectDrawSurface3::Blt** with the DDBLT_COLORFILL flag to first fill the surface with the color blue. Then you can write everything else on top of it. This allows you to fill in the most common color very quickly, and you then only have to write a minimum number of colors to the surface.

The following example demonstrates one way to perform a color fill:

```
DDBLTFX ddbltfx;

ddbltfx.dwSize = sizeof(ddbltfx);
ddbltfx.dwFillColor = 0;
ddrval = lpDDSPrimary->Blt(
    NULL,       // Destination
    NULL, NULL,  // Source rectangle
    DDBLT_COLORFILL, &ddbltfx);

switch(ddrval)
{
    case DDERR_WASSTILLDRAWING:
        .
        .
        .
    case DDERR_SURFACELOST:
        .
        .
        .
    case DD_OK:
        .
        .
        .
    default:
}
```

Determining the Capabilities of the Display Hardware

DirectDraw uses software emulation to perform the DirectDraw functions not supported by the user's hardware. To accelerate performance of your DirectDraw applications, you should determine the capabilities of the user's display hardware after you have created a DirectDraw object, then structure your program to take advantage of these capabilities when possible.

You can determine these capabilities by using the **IDirectDraw2::GetCaps** method. Not all hardware features are supported in emulation. If you want to use a feature only supported by some hardware, you must also be prepared to supply some alternative for systems with hardware that lacks that feature.

Storing Bitmaps in Display Memory

Blitting from display memory to display memory is usually much more efficient than blitting from system memory to display memory. As a result, you should store as many of the sprites your application uses as possible in display memory.

Most display adapter hardware contains enough extra memory to store more than only the primary surface and the back buffer. You can use the **dwVidMemTotal** and **dwVidMemFree** members of the **DDCAPS** structure (if you used the **IDirectDraw2::GetCaps** method to get the capabilities of the user's display hardware) to determine the amount of available memory for storing bitmaps in the display adapter's memory. If you want to see how this works, use the DirectX Viewer sample application included with the DirectX APIs in the Platform SDK. Under DirectDraw Devices, open the Primary Display Driver folder, and then open the General folder. The amount of total display memory (minus the primary surface) and the amount of free memory is displayed. Each time a surface is added to the DirectDraw object, the amount of free memory decreases by the amount of memory used by the added surface.

Triple Buffering

In some cases, that is, when the display adapter has enough memory, it may be possible to speed up the process of displaying your application by using triple buffering. Triple buffering uses one primary surface and two back buffers. The following example shows how to initialize a triple-buffering scheme:

```
// The lpDDSPrimary, lpDDSMiddle, and lpDDSBack are globally
// declared, uninitialized LPDIRECTDRAWSURFACE variables.

DDSURFACEDESC ddsd;
ZeroMemory (&ddsd, sizeof(ddsd));
```

```
// Create the primary surface with two back buffers.
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
   DDSCAPS_FLIP | DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 2;
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSPrimary, NULL);

// If we successfully created the flipping chain,
// retrieve pointers to the surfaces we need for
// flipping and blitting.
if(ddrval == DD_OK)
{
   // Get the surface directly attached to the primary (the back buffer).
   ddsd.ddsCaps.dwCaps = DDSCAPS_BACKBUFFER;
   ddrval = lpDDSPrimary->GetAttachedSurface(&ddsd.ddsCaps,
      &lpDDSMiddle);
   if(ddrval != DD_OK) ;
      // Display an error message here.
}
```

You do not need to keep track of all surfaces in a triple buffered flipping chain. The only surfaces you must keep pointers to are the primary surface and the back-buffer surface. You need a pointer to the primary surface in order to flip the surfaces in the flipping chain, and you need a pointer to the back buffer for blitting. For more information, see Flipping Surfaces.

Triple buffering allows your application to continue blitting to the back buffer even if a flip has not completed and the back buffer's blit has already finished. Performing a flip is not a synchronous event; one flip can take longer than another. Therefore, if your application uses only one back buffer, it may spend some time idling while waiting for the **IDirectDrawSurface3::Flip** method to return with DD_OK.

DirectDraw Applications and Window Styles

If your application uses DirectDraw in windowed mode, you can create windows with any window style. However, full screen, exclusive mode applications cannot be created with the WS_EX_TOOLWINDOW style without risk of unpredictable behavior. The WS_EX_TOOLWINDOW style prevents a window from being the top most window, which is required for a DirectDraw full screen, exclusive mode application.

Full screen exclusive mode applications should use the WS_EX_TOPMOST extended window style and the WS_VISIBLE window style to display properly. These styles keep the application at the front of the window z-order and prevent GDI from drawing on the primary surface.

The following example shows one way to safely prepare a window to be used in a full-screen, exclusive mode application.

```
//////////////////////////////////////////////////////
// Register the window class, display the window, and init
// all DirectX and graphic objects.
//////////////////////////////////////////////////////
BOOL WINAPI InitApp(INT nWinMode)
{
   WNDCLASSEX wcex;

   wcex.cbSize          =   sizeof(WNDCLASSEX);
   wcex.hInstance       =   g_hinst;
   wcex.lpszClassName   =   g_szWinName;
   wcex.lpfnWndProc     =   WndProc;
   wcex.style           =   CS_VREDRAW|CS_HREDRAW|CS_DBLCLKS;
   wcex.hIcon           =   LoadIcon (NULL, IDI_APPLICATION);
   wcex.hIconSm         =   LoadIcon (NULL, IDI_WINLOGO);
   wcex.hCursor         =   LoadCursor (NULL, IDC_ARROW);
   wcex.lpszMenuName    =   MAKEINTRESOURCE(IDR_APPMENU);
   wcex.cbClsExtra      =   0 ;
   wcex.cbWndExtra      =   0 ;
   wcex.hbrBackground   =   GetStockObject (NULL_BRUSH);

   RegisterClassEx(&wcex);

   g_hwndMain = CreateWindowEx(
                WS_EX_TOPMOST,
                g_szWinName,
                g_szWinCaption,
                WS_VISIBLE|WS_POPUP,
                0,0,CX_SCREEN,CY_SCREEN,
                NULL,
                NULL,
                g_hinst,
                NULL);

   if(!g_hwndMain)
      return(FALSE);

   SetFocus(g_hwndMain);
   ShowWindow(g_hwndMain, nWinMode);
   UpdateWindow(g_hwndMain);

   return TRUE;
}
```

Matching True RGB Colors to the Frame Buffer's Color Space

Applications often need to find out how a true RGB color (RGB 888) will be mapped into a the frame buffer's color space when the display device is not in RGB 888 mode. For example, imagine you're working on an application that will run in 16 and 24 bit RGB display modes. You know that when the art was created, a color was reserved for use as a transparent blitting color key; for the sake of argument, it is a 24 bit color such as RGB(128,64,255). Because your application will also run in a 16 bit RGB mode, you need a way to find out how this 24 bit color key maps into the color space that the frame buffer uses when it's running in a 16 bit RGB mode.

Although DirectDraw does not perform color matching services for you, there are ways to calculate how your color key will be mapped in the frame buffer. These methods can be pretty complicated. For most purposes, you can use the GDI built-in color matching services, combined with the DirectDraw direct frame buffer access, to determine how a color value maps into a different color space. In fact, the Ddutil.cpp source file included in the DirectX examples of the Platform SDK includes a sample function called **DDColorMatch** that performs this task. The **DDColorMatch** sample function performs the following main tasks:

1. Retrieves the color value of a pixel in a surface at 0,0.
2. Calls the Win32 **SetPixel** function, using a **COLORREF** structure that describes your 24-bit RGB color.
3. Uses DirectDraw to lock the surface, getting a pointer to the frame buffer memory.
4. Retrieves the actual color value from the frame buffer (set by GDI in Step 2) and unlocks the surface
5. Resets the pixel at 0,0 to its original color using **SetPixel**.

The process used by the **DDColorMatch** sample function is not fast; it isn't intended to be. However, it provides a reliable way to determine how a color will be mapped across different RGB color spaces. For more information, see the source code for **DDColorMatch** in the Ddutil.cpp source file.

DirectDraw Tutorials

This section contains a series of tutorials, each of which provides step-by-step instructions for implementing a simple DirectDraw application. These tutorials use many of the DirectDraw sample files that are provided with this SDK. These samples demonstrate how to set up DirectDraw, and how to use the DirectDraw methods to perform common tasks:

- Tutorial 1: The Basics of DirectDraw

- Tutorial 2: Loading Bitmaps on the Back Buffer
- Tutorial 3: Blitting from an Off-Screen Surface
- Tutorial 4: Color Keys and Bitmap Animation
- Tutorial 5: Dynamically Modifying Palettes
- Tutorial 6: Using Overlay Surfaces

Some samples in these tutorials use the older **IDirectDraw** and **IDirectDrawSurface** interfaces. If you want to update these examples so they use the DirectX 5 interfaces query for the new versions of the interfaces before using them. In addition, you must change the appropriate parameters of any methods that have been updated for new versions of the interfaces.

Note

The sample files in these tutorials are written in C++. If you are using a C compiler, you must make the appropriate changes to the files for them to successfully compile. At the very least, you need to add the *vtable* and *this* pointers to the interface methods.

Tutorial 1: The Basics of DirectDraw

To use DirectDraw, you first create an instance of the DirectDraw object, which represents the display adapter on the computer. You then use the interface methods to manipulate the object. In addition, you need to create one or more instances of a DirectDrawSurface object to be able to display your application on a graphics surface.

To demonstrate this, the DDEX1 sample included with this SDK performs the following steps:

- Step 1: Creating a DirectDraw Object
- Step 2: Determining the Application's Behavior
- Step 3: Changing the Display Mode
- Step 4: Creating Flipping Surfaces
- Step 5: Rendering to the Surfaces
- Step 6: Writing to the Surface
- Step 7: Flipping the Surfaces
- Step 8: Deallocating the DirectDraw Objects

Note:

To use GUIDs successfully in your applications, you must either define INITGUID prior to all other include and define statements, or you must link to the DXGUID.LIB library. You should define INITGUID in only one of your source modules.

Step 1: Creating a DirectDraw Object

To create an instance of a DirectDraw object, your application should use the **DirectDrawCreate** function as shown in the **doInit** function of the DDEX1 program. **DirectDrawCreate** contains three parameters. The first parameter takes a globally unique identifier (GUID) that represents the display device. The GUID, in most cases, is set to NULL, which means DirectDraw uses the default display driver for the system. The second parameter contains the address of a pointer that identifies the location of the DirectDraw object if it is created. The third parameter is always set to NULL and is included for future expansion.

The following example shows how to create the DirectDraw object and how to determine if the creation was successful or not:

```
ddrval = DirectDrawCreate(NULL, &lpDD, NULL);
if(ddrval == DD_OK)
{
   // lpDD is a valid DirectDraw object.
}
else
{
   // The DirectDraw object could not be created.
}
```

Step 2: Determining the Application's Behavior

Before you can change the resolution of your display, you must at a minimum specify the DDSCL_EXCLUSIVE and DDSCL_FULLSCREEN flags in the *dwFlags* parameter of the **IDirectDraw2::SetCooperativeLevel** method. This gives your application complete control over the display device, and no other application will be able to share it. In addition, the DDSCL_FULLSCREEN flag sets the application in exclusive (full-screen) mode. Your application covers the entire desktop, and only your application can write to the screen. The desktop is still available, however. (To see the desktop in an application running in exclusive mode, start DDEX1 and press ALT + TAB.)

The following example demonstrates the use of the **IDirectDraw2::SetCooperativeLevel** method:

```
HRESULT      ddrval;
LPDIRECTDRAW lpDD;     // Already created by DirectDrawCreate

ddrval = lpDD->SetCooperativeLevel(hwnd, DDSCL_EXCLUSIVE |
   DDSCL_FULLSCREEN);
if(ddrval == DD_OK)
{
```

```
    // Exclusive mode was successful.
}
else
{
    // Exclusive mode was not successful.
    // The application can still run, however.
}
```

If **IDirectDraw2::SetCooperativeLevel** does not return DD_OK, you can still run your application. The application will not be in exclusive mode, however, and it might not be capable of the performance your application requires. In this case, you might want to display a message that allows the user to decide whether or not to continue.

One requirement for using **IDirectDraw2::SetCooperativeLevel** is that you must pass a handle of a window (**HWND**) to allow Windows to determine if your application terminates abnormally. For example, if a general protection (GP) fault occurs and GDI is flipped to the back buffer, the user will not be able to return to the Windows screen. To prevent this from occurring, DirectDraw provides a process running in the background that traps messages that are sent to that window. DirectDraw uses these messages to determine when the application terminates. This feature imposes some restrictions, however. You have to specify the window handle that is retrieving messages for your application—that is, if you create another window, you must ensure that you specify the window that is active. Otherwise, you might experience problems, including unpredictable behavior from GDI, or no response when you press ALT+TAB.

Step 3: Changing the Display Mode

After you have set the application's behavior, you can use the **IDirectDraw2::SetDisplayMode** method to change the resolution of the display. The following example shows how to set the display mode to 640×480×8 bpp:

```
HRESULT     ddrval;
LPDIRECTDRAW lpDD;  // Already created

ddrval = lpDD->SetDisplayMode(640, 480, 8);
if(ddrval == DD_OK)
{
    // The display mode changed successfully.
}
else
{
    // The display mode cannot be changed.
    // The mode is either not supported or
    // another application has exclusive mode.
}
```

When you set the display mode, you should ensure that if the user's hardware cannot support higher resolutions, your application reverts to a standard mode that is supported by a majority of display adapters. For example, your application could be designed to run on all systems that support 640×480×8 as a standard backup resolution.

Note:

IDirectDraw2::SetDisplayMode returns a DDERR_INVALIDMODE error value if the display adapter could not be set to the desired resolution. Therefore, you should use the **IDirectDraw2::EnumDisplayModes** method to determine the capabilities of the user's display adapter before trying to set the display mode.

Step 4: Creating Flipping Surfaces

After you have set the display mode, you must create the surfaces on which to place your application. Because the DDEX1 example is using the **IDirectDraw2::SetCooperativeLevel** method to set the mode to exclusive (full-screen) mode, you can create surfaces that flip between the surfaces. If you were using **IDirectDraw2::SetCooperativeLevel** to set the mode to DDSCL_NORMAL, you could create only surfaces that blit between the surfaces. Creating flipping surfaces requires the following steps, also discussed in this topic:

- Defining the surface requirements
- Creating the surfaces

Defining the Surface Requirements

The first step in creating flipping surfaces is to define the surface requirements in a **DDSURFACEDESC** structure. The following example shows the structure definitions and flags needed to create a flipping surface.

```
// Create the primary surface with one back buffer.
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
   DDSCAPS_FLIP | DDSCAPS_COMPLEX;

ddsd.dwBackBufferCount = 1;
```

In this example, the **dwSize** member is set to the size of the **DDSURFACEDESC** structure. This is to prevent any DirectDraw method call you use from returning with an invalid member error. (The **dwSize** member was provided for future expansion of the **DDSURFACEDESC** structure.)

The **dwFlags** member determines which members in the **DDSURFACEDESC** structure will be filled with valid information. For the DDEX1 example, **dwFlags** is

set to specify that you want to use the **DDSCAPS** structure (DDSD_CAPS) and that you want to create a back buffer (DDSD_BACKBUFFERCOUNT).

The **dwCaps** member in the example indicates the flags that will be used in the **DDSCAPS** structure. In this case, it specifies a primary surface (DDSCAPS_PRIMARYSURFACE), a flipping surface (DDSCAPS_FLIP), and a complex surface (DDSCAPS_COMPLEX).

Finally, the example specifies one back buffer. The back buffer is where the backgrounds and sprites will actually be written. The back buffer is then flipped to the primary surface. In the DDEX1 example, the number of back buffers is set to 1. You can, however, create as many back buffers as the amount of display memory allows. For more information on creating more than one back buffer, see Triple Buffering.

Surface memory can be either display memory or system memory. DirectDraw uses system memory if the application runs out of display memory (for example, if you specify more than one back buffer on a display adapter with only 1 MB of RAM). You can also specify whether to use only system memory or only display memory by setting the **dwCaps** member in the **DDSCAPS** structure to DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY. (If you specify DDSCAPS_VIDEOMEMORY, but not enough memory is available to create the surface, **IDirectDraw2::CreateSurface** returns with a DDERR_OUTOFVIDEOMEMORY error.)

Creating the Surfaces

After the **DDSURFACEDESC** structure is filled, you can use it and *lpDD*, the pointer to the DirectDraw object that was created by the **DirectDrawCreate** function, to call the **IDirectDraw2::CreateSurface** method, as shown in the following example:

```
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSPrimary, NULL);
if(ddrval == DD_OK)
{
   // lpDDSPrimary points to the new surface.
}
else
{
   // The surface was not created.
   return FALSE;
}
```

The *lpDDSPrimary* parameter will point to the primary surface returned by **IDirectDraw2::CreateSurface** if the call succeeds.

After the pointer to the primary surface is available, you can use the **IDirectDrawSurface3::GetAttachedSurface** method to retrieve a pointer to the back buffer, as shown in the following example:

```
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
```

```
ddrval = lpDDSPrimary->GetAttachedSurface(&ddcaps, &lpDDSBack);
if(ddrval == DD_OK)
{
   // lpDDSBack points to the back buffer.
}
else
{
   return FALSE;
}
```

By supplying the address of the surface's primary surface and by setting the capabilities value with the DDSCAPS_BACKBUFFER flag, the *lpDDSBack* parameter will point to the back buffer if the **IDirectDrawSurface3::GetAttachedSurface** call succeeds.

Step 5: Rendering to the Surfaces

After the primary surface and a back buffer have been created, the DDEX1 example renders some text on the primary surface and back buffer surface by using standard Windows GDI functions, as shown in the following example:

```
if (lpDDSPrimary->GetDC(&hdc) == DD_OK)
{
   SetBkColor(hdc, RGB(0, 0, 255));
   SetTextColor(hdc, RGB(255, 255, 0));
   TextOut(hdc, 0, 0, szFrontMsg, lstrlen(szFrontMsg));
   lpDDSPrimary->ReleaseDC(hdc);
}

if (lpDDSBack->GetDC(&hdc) == DD_OK)
{
   SetBkColor(hdc, RGB(0, 0, 255));
   SetTextColor(hdc, RGB(255, 255, 0));
   TextOut(hdc, 0, 0, szBackMsg, lstrlen(szBackMsg));
   lpDDSBack->ReleaseDC(hdc);
}
```

The example uses the **IDirectDrawSurface3::GetDC** method to retrieve the handle of the device context, and it internally locks the surface. If you are not going to use Windows functions that require a handle of a device context, you could use the **IDirectDrawSurface3::Lock** and **IDirectDrawSurface3::Unlock** methods to lock and unlock the back buffer.

Locking the surface memory (whether the whole surface or part of a surface) ensures that your application and the system blitter cannot obtain access to the surface memory at the same time. This prevents errors from occurring while your application

is writing to surface memory. In addition, your application cannot page flip until the surface memory is unlocked.

After the surface is locked, the example uses standard Windows GDI functions: **SetBkColor** to set the background color, **SetTextColor** to select the color of the text to be placed on the background, and **TextOut** to print the text and background color on the surfaces.

After the text has been written to the buffer, the example uses the **IDirectDrawSurface3::ReleaseDC** method to unlock the surface and release the handle. Whenever your application finishes writing to the back buffer, you must call either **IDirectDrawSurface3::ReleaseDC** or **IDirectDrawSurface3::Unlock**, depending on your application. Your application cannot flip the surface until the surface is unlocked.

Typically, you write to a back buffer, which you then flip to the primary surface to be displayed. In the case of DDEX1, there is a significant delay before the first flip, so DDEX1 writes to the primary buffer in the initialization function to prevent a delay before displaying the surface. As you will see in a subsequent step of this tutorial, the DDEX1 example writes only to the back buffer during WM_TIMER. An initialization function or title page may be the only place where you might want to write to the primary surface.

Note

After the surface is unlocked by using **IDirectDrawSurface3::Unlock**, the pointer to the surface memory is invalid. You must use **IDirectDrawSurface3::Lock** again to obtain a valid pointer to the surface memory.

Step 6: Writing to the Surface

The first half of the WM_TIMER message in DDEX1 is devoted to writing to the back buffer, as shown in the following example:

```
case WM_TIMER:
   // Flip surfaces.
   if(bActive)
   {
      if (lpDDSBack->GetDC(&hdc) == DD_OK)
      {
         SetBkColor(hdc, RGB(0, 0, 255));
         SetTextColor(hdc, RGB(255, 255, 0));
         if(phase)
         {
            TextOut(hdc, 0, 0, szFrontMsg, lstrlen(szFrontMsg));
            phase = 0;
         }
         else
```

```
        {
            TextOut(hdc, 0, 0, szBackMsg, lstrlen(szBackMsg));
            phase = 1;
        }
        lpDDSBack->ReleaseDC(hdc);
    }
```

The line of code that calls the **IDirectDrawSurface3::GetDC** method locks the back buffer in preparation for writing. The **SetBkColor** and **SetTextColor** functions set the colors of the background and text.

Next, the *phase* variable determines whether the primary buffer message or the back buffer message should be written. If *phase* equals 1, the primary surface message is written, and *phase* is set to 0. If *phase* equals 0, the back buffer message is written, and *phase* is set to 1. Note, however, that in both cases the messages are written to the back buffer.

After the message is written to the back buffer, the back buffer is unlocked by using the **IDirectDrawSurface3::ReleaseDC** method.

Step 7: Flipping the Surfaces

After the surface memory is unlocked, you can use the **IDirectDrawSurface3::Flip** method to flip the back buffer to the primary surface, as shown in the following example:

```
while(1)
{
    HRESULT ddrval;
    ddrval = lpDDSPrimary->Flip(NULL, 0);
    if(ddrval == DD_OK)
    {
        break;
    }
    if(ddrval == DDERR_SURFACELOST)
    {
        ddrval = lpDDSPrimary->Restore();
        if(ddrval != DD_OK)
        {
            break;
        }
    }
    if(ddrval != DDERR_WASSTILLDRAWING)
    {
        break;
    }
}
```

In the example, **lpDDSPrimary** designates the primary surface and its associated back buffer. When **IDirectDrawSurface3::Flip** is called, the front and back surfaces are exchanged (only the pointers to the surfaces are changed; no data is actually moved). If the flip is successful and returns DD_OK, the application breaks from the while loop.

If the flip returns with a **DDERR_SURFACELOST** value, an attempt is made to restore the surface by using the **IDirectDrawSurface3::Restore** method. If the restore is successful, the application loops back to the **IDirectDrawSurface3::Flip** call and tries again. If the restore is unsuccessful, the application breaks from the while loop, and returns with an error.

Note

When you call **IDirectDrawSurface3::Flip**, the flip does not complete immediately. Rather, a flip is scheduled for the next time a vertical blank occurs on the system. If, for example, the previous flip has not occurred, **IDirectDrawSurface3::Flip** returns **DDERR_WASSTILLDRAWING**. In the example, the **IDirectDrawSurface3::Flip** call continues to loop until it returns DD_OK.

Step 8: Deallocating the DirectDraw Objects

When you press the F12 key, the DDEX1 application processes the WM_DESTROY message before exiting the application. This message calls the **finiObjects** function, which contains all of the **IUnknown::Release** calls, as shown below:

```
static void finiObjects(void)
{
    if(lpDD != NULL)
    {
        if(lpDDSPrimary != NULL)
        {
            lpDDSPrimary->Release();
            lpDDSPrimary = NULL;
        }
        lpDD->Release();
        lpDD = NULL;
    }
} // finiObjects
```

The application checks if the pointers to the DirectDraw object (*lpDD*) and the DirectDrawSurface object (*lpDDSPrimary*) are not equal to NULL. Then DDEX1 calls the **IDirectDrawSurface3::Release** method to decrease the reference count of the DirectDrawSurface object by 1. Because this brings the reference count to 0, the DirectDrawSurface object is deallocated. The DirectDrawSurface pointer is then destroyed by setting its value to NULL. Next, the application calls

IDirectDraw2::Release to decrease the reference count of the DirectDraw object to 0, deallocating the DirectDraw object. This pointer is then also destroyed by setting its value to NULL.

Tutorial 2: Loading Bitmaps on the Back Buffer

The sample discussed in this tutorial (DDEX2) expands on the DDEX1 sample that was discussed in Tutorial 1. DDEX2 includes functionality to load a bitmap file on the back buffer. This new functionality is demonstrated in the following steps:

- Step 1: Creating the Palette
- Step 2: Setting the Palette
- Step 3: Loading a Bitmap on the Back Buffer
- Step 4: Flipping the Surfaces

As in DDEX1, **doInit** is the initialization function for the DDEX2 application. Although the code for the DirectDraw initialization does not look quite the same in DDEX2 as it did in DDEX1, it is essentially the same, except for the following section:

```
lpDDPal = DDLoadPalette(lpDD, szBackground);

if (lpDDPal == NULL)
   goto error;

ddrval = lpDDSPrimary->SetPalette(lpDDPal);

if(ddrval != DD_OK)
   goto error;

// Load a bitmap into the back buffer.
ddrval = DDReLoadBitmap(lpDDSBack, szBackground);

if(ddrval != DD_OK)
   goto error;
```

Step 1: Creating the Palette

The DDEX2 sample first loads the palette into a structure by using the following code:

```
lpDDPal = DDLoadPalette(lpDD, szBackground);

if (lpDDPal == NULL)
```

```
    goto error;
```

DDLoadPalette is part of the common DirectDraw functions found in the Ddutil.cpp file located in the \Dxsdk\Sdk\Samples\Misc directory. Most of the DirectDraw sample files in this SDK use this file. Essentially, it contains the functions for loading bitmaps and palettes from either files or resources. To avoid having to repeat code in the example files, these functions were placed in a file that could be reused. Make sure you include Ddutil.cpp in the list of files to be compiled with the rest of the DDEX samples.

For DDEX2, the **DDLoadPalette** function creates a DirectDrawPalette object from the Back.bmp file. The **DDLoadPalette** function determines if a file or resource for creating a palette exists. If one does not, it creates a default palette. For DDEX2, it extracts the palette information from the bitmap file and stores it in a structure pointed to by *ape*.

DDEX2 then creates the DirectDrawPalette object, as shown in the following example:

```
pdd->CreatePalette(DDPCAPS_8BIT, ape, &ddpal, NULL);
return ddpal;
```

When the **IDirectDraw2::CreatePalette** method returns, the *ddpal* parameter points to the DirectDrawPalette object, which is then returned from the **DDLoadPalette** call.

The *ape* parameter is a pointer to a structure that can contain either 2, 4, 16, or 256 entries, organized linearly. The number of entries depends on the *dwFlags* parameter in the **IDirectDraw2::CreatePalette** method. In this case, the *dwFlags* parameter is set to DDPCAPS_8BIT, which indicates that there are 256 entries in this structure. Each entry contains 4 bytes (a red channel, a green channel, a blue channel, and a flags byte).

Step 2: Setting the Palette

After you create the palette, you pass the pointer to the DirectDrawPalette object (*ddpal*) to the primary surface by calling the **IDirectDrawSurface3::SetPalette** method, as shown in the following example:

```
ddrval = lpDDSPrimary->SetPalette(lpDDPal);

if(ddrval != DD_OK)
   // SetPalette failed.
```

After you have called **IDirectDrawSurface3::SetPalette**, the DirectDrawPalette object is associated with the DirectDrawSurface object. Any time you need to change the palette, you simply create a new palette and set the palette again. (Although this tutorial uses these steps, there are other ways of changing the palette, as will be shown in later examples.)

Step 3: Loading a Bitmap on the Back Buffer

After the DirectDrawPalette object is associated with the DirectDrawSurface object, DDEX2 loads the Back.bmp bitmap on the back buffer by using the following code:

```
// Load a bitmap into the back buffer.
ddrval = DDReLoadBitmap(lpDDSBack, szBackground);

if(ddrval != DD_OK)
   // Load failed.
```

DDReLoadBitmap is another function found in Ddutil.cpp. It loads a bitmap from a file or resource into an already existing DirectDraw surface. (You could also use **DDLoadBitmap** to create a surface and load the bitmap into that surface. For more information, see Tutorial 5: Dynamically Modifying Palettes.) For DDEX2, it loads the Back.bmp file pointed to by *szBackground* onto the back buffer pointed to by *lpDDSBack*. The **DDReLoadBitmap** function calls the **DDCopyBitmap** function to copy the file onto the back buffer and stretch it to the proper size.

The **DDCopyBitmap** function copies the bitmap into memory, and it uses the **GetObject** function to retrieve the size of the bitmap. It then uses the following code to retrieve the size of the back buffer onto which it will place the bitmap:

```
// Get the size of the surface.
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_HEIGHT | DDSD_WIDTH;
pdds->GetSurfaceDesc(&ddsd);
```

The *ddsd* value is a pointer to the **DDSURFACEDESC** structure. This structure stores the current description of the DirectDraw surface. In this case, the **DDSURFACEDESC** members describe the height and width of the surface, which are indicated by DDSD_HEIGHT and DDSD_WIDTH. The call to the **IDirectDrawSurface3::GetSurfaceDesc** method then loads the structure with the proper values. For DDEX2, the values will be 480 for the height and 640 for the width.

The **DDCopyBitmap** function locks the surface and copies the bitmap to the back buffer, stretching or compressing it as applicable by using the **StretchBlt** function, as shown below:

```
if ((hr = pdds->GetDC(&hdc)) == DD_OK)
{
   StretchBlt(hdc, 0, 0, ddsd.dwWidth, ddsd.dwHeight, hdcImage, x, y,
      dx, dy, SRCCOPY);
   pdds->ReleaseDC(hdc);
}
```

Step 4: Flipping the Surfaces

Flipping surfaces in the DDEX2 sample is essentially the same process as that in the DDEX1 tutorial (see Tutorial 1: The Basics of DirectDraw) except that if the surface is lost (**DDERR_SURFACELOST**), the bitmap must be reloaded on the back buffer by using the **DDReLoadBitmap** function after the surface is restored.

Tutorial 3: Blitting from an Off-Screen Surface

The sample in Tutorial 2 (DDEX2) takes a bitmap and puts it in the back buffer, and then it flips between the back buffer and the primary buffer. This is not a very realistic approach to displaying bitmaps. The sample in this tutorial (DDEX3) expands on the capabilities of DDEX2 by including two off-screen buffers in which the two bitmaps—one for the even screen and one for the odd screen—are stored. It uses the **IDirectDrawSurface3::BltFast** method to copy the contents of an off-screen surface to the back buffer, and then it flips the buffers and copies the next off-screen surface to the back buffer.

The new functionality demonstrated in DDEX3 is shown in the following steps:

- Step 1: Creating the Off-Screen Surfaces
- Step 2: Loading the Bitmaps to the Off-Screen Surfaces
- Step 3: Blitting the Off-Screen Surfaces to the Back Buffer

Step 1: Creating the Off-Screen Surfaces

The following code is added to the **doInit** function in DDEX3 to create the two off-screen buffers:

```
// Create an offscreen bitmap.
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 480;
ddsd.dwWidth = 640;
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSOne, NULL);
if(ddrval != DD_OK)
{
   return initFail(hwnd);
}

// Create another offscreen bitmap.
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSTwo, NULL);
if(ddrval != DD_OK)
{
```

```
    return initFail(hwnd);
}
```

The **dwFlags** member specifies that the application will use the **DDSCAPS** structure, and it will set the height and width of the buffer. The surface will be an off-screen plain buffer, as indicated by the DDSCAPS_OFFSCREEN flag set in the **DDSCAPS** structure. The height and the width are set as 480 and 640, respectively, in the **DDSURFACEDESC** structure. The surface is then created by using the **IDirectDraw2::CreateSurface** method.

Because both of the off-screen plain buffers are the same size, the only requirement for creating the second buffer is to call **IDirectDraw2::CreateSurface** again with a different pointer name.

You can also specifically request that the off-screen buffer be placed in system memory or display memory by setting either the DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY capability in the **DDSCAPS** structure. By saving the bitmaps in display memory, you can increase the speed of the transfers between the off-screen surfaces and the back buffer. This will become more important when using bitmap animation. However, if you specify DDSCAPS_VIDEOMEMORY for the off-screen buffer and not enough display memory is available to hold the entire bitmap, a DDERR_OUTOFVIDEOMEMORY error value will be returned when you attempt to create the surface.

Step 2: Loading the Bitmaps to the Off-Screen Surfaces

After the two off-screen surfaces are created, DDEX3 uses the **InitSurfaces** function to load the bitmaps from the Frntback.bmp file onto the surfaces. The **InitSurfaces** function uses the **DDCopyBitmap** function located in Ddutil.cpp to load both of the bitmaps, as shown in the following example:

```
// Load the bitmap resource.
hbm = (HBITMAP)LoadImage(GetModuleHandle(NULL), szBitmap,
    IMAGE_BITMAP, 0, 0, LR_CREATEDIBSECTION);

if (hbm == NULL)
    return FALSE;

DDCopyBitmap(lpDDSOne, hbm, 0, 0,   640, 480);
DDCopyBitmap(lpDDSTwo, hbm, 0, 480, 640, 480);
DeleteObject(hbm);

return TRUE;
```

If you look at the Frntback.bmp file in Microsoft Paint or another drawing application, you can see that the bitmap consists of two screens, one on top of the

other. The **DDCopyBitmap** function breaks the bitmap in two at the point where the screens meet. In addition, it loads the first bitmap into the first off-screen surface (*lpDDSOne*) and the second bitmap into the second off-screen surface (*lpDDSTwo*).

Step 3: Blitting the Off-Screen Surfaces to the Back Buffer

The WM_TIMER message contains the code for writing to surfaces and flipping surfaces. In the case of DDEX3, it contains the following code to select the proper off-screen surface and to blit it to the back buffer:

```
rcRect.left = 0;
rcRect.top = 0;
rcRect.right = 640;
rcRect.bottom = 480;
if(phase)
{
   pdds = lpDDSTwo;
   phase = 0;
}
else
{
   pdds = lpDDSOne;
   phase = 1;
}
while(1)
{
   ddrval = lpDDSBack->BltFast(0, 0, pdds, &rcRect, FALSE);
   if(ddrval == DD_OK)
   {
      break;
   }
```

The phase variable determines which off-screen surface will be blitted to the back buffer. The **IDirectDrawSurface3::BltFast** method is then called to blit the selected off-screen surface onto the back buffer, starting at position (0, 0), the upper-left corner. The *rcRect* parameter points to the **RECT** structure that defines the upper-left and lower-right corners of the off-screen surface that will be blitted from. The last parameter is set to FALSE (or 0), indicating that no specific transfer flags are used.

Depending on the requirements of your application, you could use either the **IDirectDrawSurface3::Blt** method or the **IDirectDrawSurface::BltFast** method to blit from the off-screen buffer. If you are performing a blit from an off-screen plain buffer that is in display memory, you should use **IDirectDrawSurface3::BltFast**. Although you will not gain speed on systems that use hardware blitter on their display adapters, the blit will take about 10 percent less time on systems that use hardware emulation to perform the blit. Because of this, you should use

IDirectDrawSurface3::BltFast for all display operations that blit from display memory to display memory. If you are blitting from system memory or require special hardware flags, however, you have to use **IDirectDrawSurface3::Blt**.

After the off-screen surface is loaded in the back buffer, the back buffer and the primary surface are flipped in much the same way as shown in the previous tutorials.

Tutorial 4: Color Keys and Bitmap Animation

The sample in Tutorial 3 (DDEX3) shows one simple method of placing bitmaps into an off-screen buffer before they are blitted to the back buffer. The sample in this tutorial (DDEX4) uses the techniques described in the previous tutorials to load a background and a series of sprites into an off-screen surface. Then it uses the **IDirectDrawSurface3::BltFast** method to copy portions of the off-screen surface to the back buffer, thereby generating a simple bitmap animation.

The bitmap file that DDEX4 uses, All.bmp, contains the background and 60 iterations of a rotating red donut with a black background. The DDEX4 sample contains new functions that set the color key for the rotating donut sprites. Then, the sample copies the appropriate sprite to the back buffer from the off-screen surface.

The new functionality demonstrated in DDEX4 is shown in the following steps:

- Step 1: Setting the Color Key
- Step 2: Creating a Simple Animation

Step 1: Setting the Color Key

In addition to the other functions found in the **doInit** function of some of the other DirectDraw samples, the DDEX4 sample contains the code to set the color key for the sprites. Color keys are used for setting a color value that will be used for transparency. When the system contains a hardware blitter, all the pixels of a rectangle are blitted except the value that was set as the color key, thereby creating nonrectangular sprites on a surface. The code for setting the color key in DDEX4 is shown below:

```
// Set the color key for this bitmap (black).
DDSetColorKey(lpDDSOne, RGB(0,0,0));

return TRUE;
```

You can select the color key by setting the RGB values for the color you want in the call to the **DDSetColorKey** function. The RGB value for black is (0, 0, 0). The **DDSetColorKey** function calls the **DDColorMatch** function. (Both functions are in Ddutil.cpp.) The **DDColorMatch** function stores the current color value of the pixel at location (0, 0) on the bitmap located in the *lpDDSOne* surface. Then it takes the

RGB values you supplied and sets the pixel at location (0, 0) to that color. Finally, it masks the value of the color with the number of bits per pixel that are available. After that is done, the original color is put back in location (0, 0), and the call returns to **DDSetColorKey** with the actual color key value. After it is returned, the color key value is placed in the **dwColorSpaceLowValue** member of the **DDCOLORKEY** structure. It is also copied to the **dwColorSpaceHighValue** member. The call to **IDirectDrawSurface3::SetColorKey** then sets the color key.

You may have noticed the reference to CLR_INVALID in **DDSetColorKey** and **DDColorMatch**. If you pass CLR_INVALID as the color key in the **DDSetColorKey** call in DDEX4, the pixel in the upper-left corner (0, 0) of the bitmap will be used as the color key. As the DDEX4 bitmap is delivered, that does not mean much because the color of the pixel at (0, 0) is a shade of gray. If, however, you would like to see how to use the pixel at (0, 0) as the color key for the DDEX4 sample, open the All.bmp bitmap file in a drawing application and then change the single pixel at (0, 0) to black. Be sure to save the change (it's hard to see). Then change the DDEX4 line that calls **DDSetColorKey** to the following:

```
DDSetColorKey(lpDDSOne, CLR_INVALID);
```

Recompile the DDEX4 sample, and ensure that the resource definition file is also recompiled so that the new bitmap is included. (To do this, you can simply add and then delete a space in the Ddex4.rc file.) The DDEX4 sample will then use the pixel at (0, 0), which is now set to black, as the color key.

Step 2: Creating a Simple Animation

The DDEX4 sample uses the **updateFrame** function to create a simple animation using the red donuts included in the All.bmp file. The animation consists of three red donuts positioned in a triangle and rotating at various speeds. This sample compares the Win32 **GetTickCount** function with the number of milliseconds since the last call to **GetTickCount** to determine whether to redraw any of the sprites. It subsequently uses the **IDirectDrawSurface3::BltFast** method first to blit the background from the off-screen surface (*lpDDSOne*) to the back buffer, and then to blit the sprites to the back buffer using the color key that you set earlier to determine which pixels are transparent. After the sprites are blitted to the back buffer, DDEX4 calls the **IDirectDrawSurface3::Flip** method to flip the back buffer and the primary surface.

Note that when you use **IDirectDrawSurface3::BltFast** to blit the background from the off-screen surface, the *dwTrans* parameter that specifies the type of transfer is set to DDBLTFAST_NOCOLORKEY. This indicates that a normal blit will occur with no transparency bits. Later, when the red donuts are blitted to the back buffer, the *dwTrans* parameter is set to DDBLTFAST_SRCCOLORKEY. This indicates that a blit will occur with the color key for transparency as it is defined, in this case, in the *lpDDSOne* buffer.

In this sample, the entire background is redrawn each time through the **updateFrame** function. One way of optimizing this sample would be to redraw only that portion of the background that changes while rotating the red donuts. Because the location and

size of the rectangles that make up the donut sprites never change, you should be able to easily modify the DDEX4 sample with this optimization.

Tutorial 5: Dynamically Modifying Palettes

The sample described in this tutorial (DDEX5) is a modification of the sample described in Tutorial 4 (DDEX4) example. DDEX5 demonstrates how to dynamically change the palette entries while an application is running. The new functionality demonstrated in DDEX5 is shown in the following steps:

- Step 1: Loading the Palette Entries
- Step 2: Rotating the Palettes

Step 1: Loading the Palette Entries

The following code in DDEX5 loads the palette entries with the values in the lower half of the All.bmp file (the part of the bitmap that contains the red donuts):

```
// First, set all colors as unused.
for(i=0; i<256; i++)
{
    torusColors[i] = 0;
}

// Lock the surface and scan the lower part (the torus area),
// and keep track of all the indexes found.
ddsd.dwSize = sizeof(ddsd);
while (lpDDSOne->Lock(NULL, &ddsd, 0, NULL) == DDERR_WASSTILLDRAWING)
    ;

// Search through the torus frames and mark used colors.
for(y=480; y<480+384; y++)
{
    for(x=0; x<640; x++)
    {
        torusColors[((BYTE *)ddsd.lpSurface)[y*ddsd.lPitch+x]] = 1;
    }
}

lpDDSOne->Unlock(NULL);
```

The **torusColors** array is used as an indicator of the color index of the palette used in the lower half of the All.bmp file. Before it is used, all of the values in the

torusColors array are reset to 0. The off-screen buffer is then locked in preparation for determining if a color index value is used.

The **torusColors** array is set to start at row 480 and column 0 of the bitmap. The color index value in the array is determined by the byte of data at the location in memory where the bitmap surface is located. This location is determined by the **lpSurface** member of the **DDSURFACEDESC** structure, which is pointing to the memory location corresponding to row 480 and column 0 of the bitmap ($y \times$ **lPitch** + x). The location of the specific color index value is then set to 1. The y value (row) is multiplied by the **lPitch** value (found in the **DDSURFACEDESC** structure) to get the actual location of the pixel in linear memory.

The color index values that are set in **torusColors** will be used later to determine which colors in the palette are rotated. Because there are no common colors between the background and the red donuts, only those colors associated with the red donuts are rotated. If you want to check whether this is true or not, just remove the "*ddsd.lPitch" from the array and see what happens when you recompile and run the program. (Without multiplying y×**lPitch**, the red donuts are never reached and only the colors found in the background are indexed and later rotated.) For more information about width and pitch, see Width and Pitch.

Step 2: Rotating the Palettes

The **updateFrame** function in DDEX5 works in much the same way as it did in Tutorial 4 (DDEX4). It first blits the background into the back buffer, and then it blits the three donuts in the foreground. However, before it flips the surfaces, **updateFrame** changes the palette of the primary surface from the palette index that was created in the **doInit** function, as shown in the following code:

```
// Change the palette.
if(lpDDPal->GetEntries(0, 0, 256, pe) != DD_OK)
{
   return;
}

for(i=1; i<256; i++)
{
   if(!torusColors[i])
   {
      continue;
   }
   pe[i].peRed = (pe[i].peRed+2) % 256;
   pe[i].peGreen = (pe[i].peGreen+1) % 256;
   pe[i].peBlue = (pe[i].peBlue+3) % 256;
}

if(lpDDPal->SetEntries(0, 0, 256, pe) != DD_OK)
{
```

```
    return;
}
```

The **IDirectDrawPalette::GetEntries** method in the first line queries palette values from a DirectDrawPalette object. Because the palette entry values pointed to by *pe* should be valid, the method will return DD_OK and continue. The loop that follows checks **torusColors** to determine if the color index was set to 1 during its initialization. If so, the red, green, and blue values in the palette entry pointed to by *pe* are rotated.

After all of the marked palette entries are rotated, the **IDirectDrawPalette::SetEntries** method is called to change the entries in the DirectDrawPalette object. This change takes place immediately if you are working with a palette set to the primary surface.

With this done, the surfaces are subsequently flipped.

Tutorial 6: Using Overlay Surfaces

This tutorial shows you, step by step, how to use DirectDraw and hardware supported overlay surfaces in your applications. The tutorial is written around the Mosquito sample application included with the DirectX SDK samples. The Mosquito sample is a simple application that uses a flipping chain of overlay surfaces to display an animated bitmap on the desktop without blitting to the primary surface. The sample adjusts the characteristics of the overlay surface as needed to accommodate for hardware limitations.

The Mosquito sample application performs the following steps (complex tasks are divided into smaller sub-steps):

- Step 1: Creating a Primary Surface
- Step 2: Testing for Hardware Overlay Support
- Step 3: Creating an Overlay Surface
- Step 4: Displaying the Overlay Surface
- Step 5: Updating the Overlay Display Position
- Step 6: Hiding the Overlay Surface

Step 1: Creating a Primary Surface

To prepare for using overlay surfaces, you must first initialize DirectDraw and create a primary surface over which the overlay surface will be displayed. Mosquito creates a primary surface with the following code:

```
// Zero-out the structure and set the dwSize member.
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
```

```
    // Set flags and create a primary surface.
    ddsd.dwFlags = DDSD_CAPS;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
    ddrval = g_lpdd->CreateSurface(&ddsd, &g_lpddsPrimary, NULL );
```

The preceding example begins by initializing the **DDSURFACEDESC** structure it will use. It then sets the flags appropriate to create a primary surface and creates it by calling the **IDirectDraw2::CreateSurface** method. For the call, the first parameter is a pointer to a **DDSURFACEDESC** structure that describes the surface to be created. The second parameter is a pointer to a variable that will receive an **IDirectDrawSurface** interface pointer if the call succeeds. The last parameter is set to NULL to indicate that no COM aggregation is taking place.

Step 2: Testing for Hardware Overlay Support

After initializing DirectDraw, you need to verify that the device supports overlay surfaces. Because DirectDraw doesn't emulate overlays, if the hardware device driver doesn't support them, you can't continue. You can test for overlay support by retrieving the device driver capabilities with the **IDirectDraw2::GetCaps** method. After the call, look for the presence of the DDCAPS_OVERLAY flag in the **dwFlags** member of the associated **DDCAPS** structure. If the flag is present, then the display hardware supports overlays; if not, you can't use overlay surfaces with that device.

The following example, taken from the Mosquito sample application, shows how to test for hardware overlay support.

```
BOOL AreOverlaysSupported()
{
    DDCAPS  capsDrv;
    HRESULT ddrval;

    // Get driver capabilities to determine Overlay support.
    ZeroMemory(&capsDrv, sizeof(capsDrv));
    capsDrv.dwSize = sizeof(capsDrv);

    ddrval = g_lpdd->GetCaps(&capsDrv, NULL);
    if (FAILED(ddrval))
        return FALSE;

    // Does the driver support overlays in the current mode?
    // (Currently the DirectDraw emulation layer does not support overlays.
    // Overlay related APIs will fail without hardware support).
    if (!(capsDrv.dwCaps & DDCAPS_OVERLAY))
        return FALSE;

    return TRUE;
```

```
}
```

The preceding example calls the **IDirectDraw2::GetCaps** method to retrieve device driver capabilities. The first parameter for the call is the address of a **DDCAPS** that will be filled with information describing the device driver's capabilities. Because the application doesn't need information about emulation capabilities, the second parameter is set to NULL.

After retrieving the driver capabilities, the example checks the **dwCaps** member for the presence of the DDCAPS_OVERLAY flag using a logical **AND** operation. If the flag isn't present, the example returns FALSE to indicate failure. Otherwise, the example returns TRUE to indicate that the device driver supports overlay surfaces.

In your code, this might be a good time for you to check the **dwMaxVisibleOverlays** and **dwCurrentVisibleOverlays** members in the DDCAPS structure to ensure that no other overlay surfaces are in use by other applications.

Step 3: Creating an Overlay Surface

Now that you know that the driver supports overlay surfaces, you can try to create one. Because there is no standard dictating how devices must support overlay surfaces, you can't count on being able to create overlays of any particular size or pixel format. Additionally, you can't expect to succeed in creating an overlay surface on the first try. Therefore, be prepared to attempt creation multiple times starting with the most desirable characteristics, falling back on less desirable (but possibly less hardware intensive) configurations until one works.

(You can call the **IDirectDraw2::GetFourCCCodes** method to retrieve a list of FOURCC codes that describe non-RGB pixel formats that the driver will likely support for overlay surfaces. However, in you want to try using RGB overlay surfaces, it is recommended that you attempt to creating surfaces in various common RGB formats, falling back on another format if you fail.)

The Mosquito sample follows a "best case to worst case" philosophy when creating an overlay surface. Mosquito first tries to create a triple-buffered page flipping complex overlay surface. If the creation attempt fails, the sample tries the configuration with other common pixel formats. The following code fragment shows how this can be done:

```
ZeroMemory(&ddsdOverlay, sizeof(ddsdOverlay));
ddsdOverlay.dwSize = sizeof(ddsdOverlay);

ddsdOverlay.dwFlags= DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH |
            DDSD_BACKBUFFERCOUNT| DDSD_PIXELFORMAT;
ddsdOverlay.ddsCaps.dwCaps = DDSCAPS_OVERLAY | DDSCAPS_FLIP |
                DDSCAPS_COMPLEX | DDSCAPS_VIDEOMEMORY;
ddsdOverlay.dwWidth  =320;
ddsdOverlay.dwHeight =240;
ddsdOverlay.dwBackBufferCount=2;
```

```
    // Try to create an overlay surface using one of the pixel formats in our
    // global list.
    i=0;
    do{
      ddsdOverlay.ddpfPixelFormat=g_ddpfOverlayFormats[i];
      // Try to create the overlay surface
      ddrval = g_lpdd->CreateSurface(&ddsdOverlay, &g_lpddsOverlay, NULL);
    } while( FAILED(ddrval) && (++i < NUM_OVERLAY_FORMATS) );
```

The preceding example sets the flags and values within a **DDSURFACEDESC** structure to reflect a triple-buffered page flipping complex overlay surface. Then, the sample performs a loop during which it attempts to create the requested surface in a variety of common pixel formats, in order of most desirable to least desirable pixel formats. If the attempt succeeds, the loop ends. If all the attempts fail, it's likely that the display hardware doesn't have enough memory to support a triple-buffered scheme or that it doesn't support flipping overlay surfaces. In this case, the sample falls back on a less desirable configuration using a single non-flipping overlay surface, as shown in the following example:

```
    // If we failed to create a triple buffered complex overlay surface, try
    // again with a single non-flippable buffer.
    if(FAILED(ddrval))
    {
       ddsdOverlay.dwBackBufferCount=0;
       ddsdOverlay.ddsCaps.dwCaps=DDSCAPS_OVERLAY | DDSCAPS_VIDEOMEMORY;
       ddsdOverlay.dwFlags= DDSD_CAPS|DDSD_HEIGHT|DDSD_WIDTH|
DDSD_PIXELFORMAT;

       // Try to create the overlay surface
       ddrval = g_lpdd->CreateSurface(&ddsdOverlay, &g_lpddsOverlay, NULL);
       i=0;
       do{
          ddsdOverlay.ddpfPixelFormat=g_ddpfOverlayFormats[i];
          ddrval = g_lpdd->CreateSurface(&ddsdOverlay, &g_lpddsOverlay, NULL);
       } while( FAILED(ddrval) && (++i < NUM_OVERLAY_FORMATS) );

       // We couldn't create an overlay surface.  Exit, returning failure.
       if (FAILED(ddrval))
          return FALSE;
    }
```

The code above resets the flags and values in the **DDSURFACEDESC** structure to reflect a single non-flipping overlay surface. Again, the example loops through pixel formats attempting to create the surfaces, stopping the loop if an attempt succeeded. If the attempts still didn't work, the sample returns FALSE to indicate failure.

After you've successfully created your overlay surface or surfaces, you can load bitmaps onto them in preparation for display.

Step 4: Displaying the Overlay Surface

After creating your overlay surface, you can display it. Often, display hardware imposes alignment restrictions on the position and pixel width of the rectangles you use to display the overlay. Additionally, you will often need to account for a minimum required stretch factor by adjusting the width of the destination rectangle in order to successfully display the overlay surface. The Mosquito sample performs the following tasks to prepare and display the overlay surface:

- Step 4.1: Determining the Minimum Display Requirements
- Step 4.2: Setting Up the Source and Destination Rectangles
- Step 4.3: Displaying the Overlay Surface

Step 4.1: Determining the Minimum Display Requirements

Most display hardware imposes restrictions on displaying overlay surfaces. You must carefully meet these restrictions in order to successfully display an overlay surface. You can retrieve information about these restrictions by calling the **IDirectDraw2::GetCaps** method. The **DDCAPS** structure that the method fills contains information about overlay capabilities and their usage restrictions. Hardware restrictions vary, so always look at the flags included in the **dwFlags** member to determine which restrictions apply to you.

The Mosquito sample starts by retrieving the hardware capabilities, then takes action based upon the minimum stretch factor, as shown in the following code fragment:

```
    // Get driver capabilities
    ddrval = g_lpdd->GetCaps(&capsDrv, NULL);
    if (FAILED(ddrval))
        return FALSE;

    // Check the minimum stretch and set the local variable accordingly.
    if(capsDrv.dwCaps & DDCAPS_OVERLAYSTRETCH)
        uStretchFactor1000 = (capsDrv.dwMinOverlayStretch>1000) ?
capsDrv.dwMinOverlayStretch : 1000;
    else
        uStretchFactor1000 = 1000;
```

The code above calls **IDirectDraw2::GetCaps** to retrieve only the hardware capabilities. For this call, the first parameter is a pointer the **DDCAPS** structure that will be filled with the capability information for the device driver, and the second parameter is NULL to indicate that emulation information is not to be retrieved.

The example retains the minimum stretch factor in a temporary variable for use later. (Keep in mind that stretch factors are reported multiplied by 1000, so 1300 really

means 1.3.) If the driver reports a value greater than 1000, it means that the driver requires that all destination rectangles must be stretched along the X-axis by a ratio of the reported value. For example, if the driver reports a stretch factor 1.3 and the source rectangle is 320 pixels wide, the destination rectangle must be at least 416 pixels wide. If the driver reports a stretch factor less than 1000, it means that the driver can display overlays smaller than the source rectangle, but can also stretch the overlay if desired.

Next, the sample examines values describing the driver's size alignment restrictions, as shown in the following example:

```
    // Grab any alignment restrictions and set the local variables acordingly.
    uSrcSizeAlign = (capsDrv.dwCaps & DDCAPS_ALIGNSIZESRC)?
capsDrv.dwAlignSizeSrc:0;
    uDestSizeAlign= (capsDrv.dwCaps & DDCAPS_ALIGNSIZESRC)?
capsDrv.dwAlignSizeDest:0;
```

The sample uses more temporary variables to hold the reported size alignment restrictions taken from the **dwAlignSizeSrc** and **dwAlignSizeDest** members. These values provide information about pixel width alignment restrictions and are needed when setting the dimensions of the source and destination rectangles to reflect these restrictions later. Source and destination rectangles must have a pixel width that is a multiple of the values in these members.

Last, the sample examines the value that describes the destination rectangle boundary alignment:

```
    // Set the "destination position alignment" global so we won't have to
    // keep calling GetCaps() every time we move the overlay surface.
    if (capsDrv.dwCaps & DDCAPS_ALIGNBOUNDARYDEST)
        g_dwOverlayXPositionAlignment = capsDrv.dwAlignBoundaryDest;
    else
        g_dwOverlayXPositionAlignment = 0;
```

The preceding code uses a global variable to hold the value for the destination rectangle's boundary alignment, as taken from the **dwAlignBoundaryDest** member. This value will be used when the program repositions the overlay later. (For details, see Step 5: Updating the Overlay Display Position) You must set the x-coordinate of the destination rectangle's top left corner to be aligned with this value, in pixels. That is, if the value specified is 4, you can only specify destination rectangles whose top-left corner has an x-coordinate at pixels 0, 4, 8, 12, and so on. The Mosquito application initially displays the overlay at 0,0, so alignment compliance is assumed and the sample doesn't need to retrieve the restriction information until after displaying the overlay the first time. Your implementation might vary, so you will probably need to check this information and adjust the destination rectangle before displaying the overlay.

Step 4.2: Setting Up the Source and Destination Rectangles

After retrieving the driver's overlay restrictions you should set the values for your source and destination rectangles accordingly, assuring that you will be able to successfully display the overlay. The following sample from the Mosquito sample application starts by setting the characteristics of the source rectangle:

```
// Set initial values in the source RECT.
rs.left=0; rs.top=0;
rs.right = 320;
rs.bottom = 240;

// Apply size alignment restrictions, if necessary.
if (capsDrv.dwCaps & DDCAPS_ALIGNSIZESRC && uSrcSizeAlign)
   rs.right -= rs.right % uSrcSizeAlign;
```

The preceding code sets initial values for the surface to include the dimensions of the entire surface. If the device driver requires size alignment for the source rectangle, the example adjusts the source rectangle to conform. The example adjusts the width of the source rectangle to be narrower than the original size because the width cannot be expanded without completely recreating the surface. However, your code could just as easily start with a smaller rectangle and widen the rectangle to meet driver restrictions.

After the dimensions of the source rectangle are set and conform with hardware restrictions, you need to set and adjust the dimensions of the destination rectangle. This process requires a little more work because the rectangle might need to be stretched first, then adjusted to meet size alignment restrictions. The following code performs the task of accounting for the minimum stretch factor:

```
// Set up the destination RECT, starting with the source RECT values.
// We use the source RECT dimensions instead of the surface dimensions in
// case they differ.
rd.left=0; rd.top=0;
rd.right  = (rs.right*uStretchFactor1000+999)/1000;      // (Adding 999 avoids integer
truncation problems.)

// (This isn't required by DDraw, but we'll stretch the
//  height, too, to maintain aspect ratio).
rd.bottom = rs.bottom*uStretchFactor1000/1000;
```

The preceding code sets the top left corner of the destination rectangle to the top left corner of the screen, then sets the width to account for the minimum stretch factor. While adjusting for the stretch factor, note that the example adds 999 to the product of the width and stretch factor. This is done to prevent integer truncation that could result in a rectangle that isn't as wide as the minimum stretch factor requires. For more information, see Minimum and Maximum Stretch Factors. Also, after the

example stretches the width, it stretches the height. Stretching the height isn't required, but was done to preserve the bitmap's aspect ratio and avoid a distorted appearance.

After stretching the destination rectangle, the example continues by adjusting it to conform to size alignment restrictions as follows:

```
// Adjust the destination RECT's width to comply with any imposed
// alignment restrictions.
if (capsDrv.dwCaps & DDCAPS_ALIGNSIZEDEST && uDestSizeAlign)
   rd.right = (int)((rd.right+uDestSizeAlign-1)/uDestSizeAlign)*uDestSizeAlign;
```

The example checks the capabilities flags to see if the driver imposes destination size alignment restrictions. If so, the destination rectangle's width is increased by enough pixels to meet alignment restrictions. Note that the rectangle is adjusted by expanding the width, not by decreasing it. This is done because decreasing the width could cause the destination rectangle to be smaller than is required by the minimum stretch factor, consequently causing attempts to display the overlay surface to fail.

Step 4.3: Displaying the Overlay Surface

After you've set up the source and destination rectangles, you can display the overlay for the first time. If you've prepared correctly, this will be simple. The Mosquito sample uses the following code to initially display the overlay:

```
// Set the flags we'll send to UpdateOverlay
dwUpdateFlags = DDOVER_SHOW | DDOVER_DDFX;

// Does the overlay hardware support source color keying?
// If so, we can hide the black background around the image.
// This probably won't work with YUV formats
if (capsDrv.dwCKeyCaps & DDCKEYCAPS_SRCOVERLAY)
   dwUpdateFlags |= DDOVER_KEYSRCOVERRIDE;

// Create an overlay FX structure so we can specify a source color key.
// This information is ignored if the DDOVER_SRCKEYOVERRIDE flag isn't set.
ZeroMemory(&ovfx, sizeof(ovfx));
ovfx.dwSize = sizeof(ovfx);

ovfx.dckSrcColorkey.dwColorSpaceLowValue=0; // Specify black as the color key
ovfx.dckSrcColorkey.dwColorSpaceHighValue=0;

// Call UpdateOverlay() to displays the overlay on the screen.
ddrval = g_lpddsOverlay->UpdateOverlay(&rs, g_lpddsPrimary, &rd, dwUpdateFlags, &ovfx);
if(FAILED(ddrval))
   return FALSE;
```

The preceding example starts by setting the DDOVER_SHOW and DDOVER_DDFX flags in the *dwUpdateFlags* temporary variable, indicating that the overlay is to be displayed for the first time, and that the hardware should use the effects information included in an associated **DDOVERLAYFX** structure to do so. Next, the example checks a previously existing **DDCAPS** structure to determine if overlay source color keying is supported. If it is, the DDOVER_KEYSRCOVERRIDE is included in the *dwUpdateFlags* variable to take advantage of source color keying and the example sets color key values accordingly.

After preparation is complete, the example calls the **IDirectDrawSurface3::UpdateOverlay** method to display the overlay. For the call, the first and third parameters are the addresses of the adjusted source and destination rectangles. The second parameter is the address of the primary surface over which the overlay will be displayed. The fourth parameter consists of the flags placed in the previously prepared *dwUpdateFlags* variable, and the fifth parameter is the address of **DDOVERLAYFX** structure whose members were set to match those flags.

If the hardware only supports one overlay surface and that surface is in use, the **UpdateOverlay** method fails, returning DDERR_OUTOFCAPS. Additionally, if **UpdateOverlay** fails, you might try increasing the width of the destination rectangle to accommodate for the possibility that the hardware incorrectly reported a minimum stretch factor that was too small. However, this rarely occurs and Mosquito simply fails if **UpdateOverlay** doesn't succeed.

Step 5: Updating the Overlay Display Position

After displaying the overlay surface, you might not need to do anything else. However, some software might need to reposition the overlay surface. The Mosquito sample uses the **IDirectDrawSurface3::SetOverlayPosition** method to reposition the overlay, as shown in the following example.

```
// Set X- and Y-coordinates
.
.
.
// We need to check for any alignment restrictions on the X position
// and align it if necessary.
if (g_dwOverlayXPositionAlignment)
   dwXAligned = g_nOverlayXPos - g_nOverlayXPos % g_dwOverlayXPositionAlignment;
else
   dwXAligned = g_nOverlayXPos;

// Set the overlay to its new position.
ddrval = g_lpddsOverlay->SetOverlayPosition(dwXAligned, g_nOverlayYPos);
if (ddrval == DDERR_SURFACELOST)
{
   if (!RestoreAllSurfaces())
      return;
```

```
    }
```

The preceding example starts by aligning the rectangle to meet any destination rectangle boundary alignment restrictions that might exist. The global variable that it checks, *g_dwOverlayXPositionAlignment*, was set earlier to equal the value reported in the **dwAlignBoundaryDest** member of the **DDCAPS** structure when the application previously called the **IDirectDraw2::GetCaps** method. (For details, see Step 4.1: Determining the Minimum Display Requirements). If destination alignment restrictions exist, the example adjusts the new x-coordinate to be pixel-aligned accordingly. Failing to meet this requirement will cause the overlay surface not to be displayed.

After making any requisite adjustments to the new x-coordinate, the example calls **IDirectDrawSurface3::SetOverlayPosition** method to reposition the overlay. For the call, the first parameter is the aligned x-coordinate, and the second parameter is the new y-coordinate. These values represent the new location of the overlay's top-left corner. Width and height information are not accepted, nor are they needed because DirectDraw already knows the dimensions of the surface from the **IDirectDrawSurface3::UpdateOverlay** method made to initially display the overlay. If the call fails because one or more surfaces were lost, the example calls an application-defined function to restore them and reload their bitmaps.

Note:

Take care not to use coordinates too close to the bottom or right edge of the target surface. The **IDirectDraw2::SetOverlayPosition** method does not perform clipping for you; using coordinates that would potentially make the overlay run off the edge of the target surface will cause the method to fail, returning DDERR_INVALIDPOSITION.

Step 6: Hiding the Overlay Surface

When you do not need the overlay surface anymore, or if you simply want to remove it from view, you can hide the surface by calling the **IDirectDrawSurface3::UpdateOverlay** method with appropriate flags. Mosquito hides the overlay in preparation for closing the application using the following code:

```
void DestroyOverlay()
{
    if (g_lpddsOverlay){
        // Use UpdateOverlay() with the DDOVER_HIDE flag to remove an overlay
        // from the display.
        g_lpddsOverlay->UpdateOverlay(NULL, g_lpddsPrimary, NULL, DDOVER_HIDE, NULL);
        g_lpddsOverlay->Release();
        g_lpddsOverlay=NULL;
    }
}
```

When the preceding example calls **IDirectDrawSurface3::UpdateOverlay**, it specifies NULL for the source and destination rectangles, because they are irrelevant when hiding the overlay. Similarly, the example uses NULL in the fourth parameter because overlay effects aren't being used. The second parameter is a pointer to the target surface. Lastly, the example uses the DDOVER_HIDE flag in the fourth parameter to indicate that the overlay will be removed from view.

After the example hides the overlay, the example releases its **IDirectDrawSurface3** interface and invalidates its global variable by setting it to NULL. For the purposes of the Mosquito sample application, the overlay surface is no longer needed. If you still need the overlay surface for later, you could simply hide the overlay without releasing it, then redisplay it whenever you require.

Other DirectDraw Samples

To learn more about how DirectDraw can be used in applications, you should check out some of the other following samples included with the DirectX SDK:

- Stretch

  Demonstrates how to create a nonexclusive (windowed) mode animation in a window that is capable of clipped blitting and stretched-clipped blitting.
- Donut

  Demonstrates testing multiple exclusive-mode applications interacting with nonexclusive-mode applications.
- Wormhole

  Demonstrates palette animation.
- Dxview

  Demonstrates how to retrieve the capabilities of the display hardware.

Other samples you can examine for their DirectDraw code include Duel, Iklowns, Foxbear, Palette, and Flip2d.