Animation in Windows

By: Herman Rodent, Nell, and Olivia the Wonder Dog

Abstract

This article is aimed at people who would like to create a Windows application which does some form of animation or who would like to understand how to improve the performance of an existing application. The main focus of the article is on using DIBs for the images and the DIB driver (DIB.DRV) for the off-screen image buffer. The article is written around a sample application (SPRITES) which is included with the article. Some knowledge of animation techniques is assumed. If you're looking for a "How to do animation" article this isn't it. The following points are covered:

- Using DIBs
- Using the DIB driver
- Palettes
- Measuring and Improving Performance
- Lots of Useful Little Tips and Hints

Introduction

A common misconception is that Windows is too slow to do any sort of animation effectively. This concept has hindered the porting of many good MS-DOS applications (notably games) to Windows. In addition to the theory that Windows is too slow, many MS-DOS programmers fear having to learn how to program in the Windows environment and recreate an existing application from scratch using strange new function calls and techniques. To top this off, there are a number of existing applications which while adequate at what they do, do not show what the system is capable of.

The Video for Windows technology recently released to developers shows what really can be achieved - reasonable quality video in a window. The frame rates are not excessive and the window sizes are not huge but the net effect is very impressive and not difficult to achieve.

If you are considering porting an MS-DOS application to Windows or would like to add some animation to an existing Windows application, then this note and its accompanying sample code should help you along the way.

For myself, I do not even pretend to be an animation wizard. Before I wrote this article I thought I had some pretty good ideas about how all this should be done and a few of them were even right! By experimenting with my own ideas and those of others I have discovered a lot of useful tips which I thought were worth passing on. Much of the information about palettes is reproduced form Ron Gery's articles available elsewhere on the Developer Network CD. Huge volumes of information also came from Todd Laney, without whom frame animation in Windows might never have got beyond one frame per second.

Architecture

When I decided to write the SPRITES sample application, I made two major choices which determined the architecture of the entire application and hence this article. These were to use DIBs for all the images and to use the DIB driver for the off-screen image buffer. Since these things are so fundamental to what follows some discussion of these two decisions is in order. This section also includes some notes on the use of palettes.

Using DIBs

Before the advent of Device Independent Bitmaps (DIBs) in Windows 3.0, we had to create applications with different DDBs (Device Dependent Bitmaps) for every screen resolution/color depth we wanted the application to run on. DIBs provide a way around that problem by packaging together information about the image size, a color table and the bits themselves into a single file. DIBs are inherently portable between different Windows environments. The only problem being that there are no Windows functions to render them directly which makes them seem somewhat difficult to use. I chose to use them in my sample application because of the portability they provide. It turns out that the code required to manipulate them is not complicated and the difficulty in writing the code is easily outweighed by the advantages of using DIBs.

DIB File Formats

One irritating complexity of the DIB file format is that there are two different versions of it. One was designed for Windows 3.0 and the other for the Presentation Manger in OS/2. The two formats essentially do the same thing but in slightly different ways.

The Windows format uses four bytes for each color table entry which keeps the color table elements DWORD aligned and is efficient to access on a 32 bit platform. The Presentation Manager format uses three bytes for each color table entry resulting in a smaller color table (big deal compared with the image size!) but inefficient access.

Both formats exist in applications designed for Windows. The Windows system handles both formats internally but may not handle the Presentation Manager format in the future. A good application which reads DIB files should be able to handle both, the only complexity being creating a palette from the color table. More about recognizing which format a DIB is in and dealing with it later.

The other irritation of the DIB format which is a hangover from the Presentation Manager design is that the scan lines in the DIB are 'upside down' with respect to their address in the file. In other words the bits for the last scan line are first and the bits for the first scan line are last. This requires some mental juggling when you manipulate or blt the bits in your code. Figure 1 shows the problem.

µ §
Figure 1. DIB scan lines are inverted.

The DIB file structure consists of two blocks: a header and the bits. The header is actually three structures packed together so the overall picture looks like figure 2.

µ §
Figure 2. Windows and Presentation Manager DIB File Formats

Note: In the case of the Windows format the BITMAPINFOHEADER and RGBQUAD array are both contained within a BITMAPINFO structure. In the case of the PM format file, the BITMAPCOREHEADER and RGBTRIPLE array are contained within a BITMAPCOREINFO structure. All of these structures are documented in the Windows SDK.

DIBs in memory

If you examine the file format structures carefully, you will see that the file header contains a pointer to the position of the image bits within the file implying that the block containing the bits need not be contiguous with the header. This is a convenience for the file format but a nuisance for a memory image format since we don't want to have to manage two memory blocks for the DIB and we don't want to allocate memory which doesn't contain any useful information. So when we read a DIB file into memory we calculate the total memory required for the header (excluding the BITMAPFILEHEADER which is not required) and the bits, allocate a single block for the whole thing and read the header and bits

separately into this block so that they end up contiguous within it.

But what about the two different formats? Instead of dealing with both formats in the application, I chose always to convert Presentation Manager DIBs to Windows DIB format internally giving only one format to deal with. This is easily done when the DIB image is created in memory. This does mean that a DIB which is subsequently saved to file later may end up being converted from Presentation Manager format to Windows format but this is not a problem for any Windows application :).

A Windows format DIB which is contained in a single memory block with its header immediately preceding the bits is called a packed DIB and is used to transfer DIBs through the Clipboard. Its Clipboard format name is CF_DIB.

Handles or Pointers?

Since we will allocate a block of memory for the packed DIB information, should we retain the handle to the memory or a pointer to it? In days of old when knights were bold and memory wasn't addressed by descriptor tables (real mode) we always worked with handles. Since we now live in a protected mode environment and are rapidly approaching the happy days of 32 bit, flat model Windows programming I have chosen to keep a pointer to all my global objects rather than a handle. This avoids thousands of unnecessary calls to GlobalLock and GlobalUnlock which both simplifies and speeds up the code. To make the code as portable as possible I use macros (ALLOCATE() and FREE()) to allocate and free memory blocks.

When I was writing the sample application I found that I needed to use things like the height and width of the DIB a lot so I initially created a structure which held all the information about the DIB I used a lot and a pointer to the packed DIB itself. This wasn't really as efficient as it might have been since we were back to having two memory blocks to describe one DIB. I decided to implement a set of macros to do all the pointer dereferencing (DIB_HEIGHT etc.) and these were used throughout the application. There is some issue here regarding the code speed. It could be argued that all the pointer dereferencing used in the macros leads to slower code than could be achieved by having the commonly used DIB parameters cached in a single place. The macros helped make the code a bit simpler to look at and resulted in a DIB being just a single memory block in CF_DIB format which on balance I prefer. Figure 3 shows the format of a packed DIB in memory.

µ §

Figure 3. Packed DIB memory organization

General DIB Notes

When accessing the bits of a DIB in memory be very careful that you use the correct type of pointer. If the DIB is less than 64k in size, you can use a FAR pointer to access all of it. If the DIB is larger than 64k then it's very important that you use a HUGE pointer so that the address arithmetic will be performed correctly. Bitmaps of 100x640, 200x320 or 250x250 are all just under 64k.

FAR pointers only have 64k offsets so if you attempt to go beyond 64k the offset will wrap and the address will have a low offset value rather than the big one you expect. This problem is most noticeable when writing to the DIB memory since an address wrap while writing the DIB bits will cause the BITMAPINFO header to be overwritten and consequently the integrity of the DIB trashed. I know about this - I've done it. The code always looks just fine , but something keeps trashing the header. Be warned!

When accessing the bits in a DIB, also be aware that the width of a scan line is always divisible by 4 (DWORD aligned) and consequently the pixel position is not simply determined by the scan line number and image width. The SPRITES header file GLOBAL.H includes a set of macros for accessing information on a DIB given a pointer to it. The DIB_STORAGEWIDTH macro returns the physical length of the scan line. The DIB_WIDTH macro returns the width of the image.

The DIB Driver

The DIB driver was developed in response to very many requests from software developers to be able to manipulate the bits of a bitmap directly in memory. The DIB driver was originally developed as a part of the Multimedia Extensions to Windows and subsequently shipped as a standard component of Windows 3.1.

Direct Bitmap Memory Access

It is not possible to directly access the bits of a Windows DDB. This is not simply because we (Microsoft) won't tell you how to find it, it's because allocation of the bitmap memory is done by the video device driver and the driver can choose to use any memory available to it including memory on the video adapter card. In the case of the 8514 driver, there is quite a bit of spare video memory in some video modes and it makes good sense for the video driver to make use of it if it can. If the memory for a bitmap is allocated on the adapter card then there is no direct access to it possible since that memory is not mapped into the processor address space.

In addition to the problem of where the memory is located, there is an additional

problem that the format of the memory allocated for the bitmap is also defined by the device driver. This makes a lot of sense since it allows the driver to choose a memory organization which makes it easy to transfer (blt) chunks to and from the video memory. Since this format can vary from driver to driver, even if you could access the memory, there is no way you could know how it was organized.

The DIB driver solves the problem of memory access and bitmap organization by requiring the application to allocate the memory for the bits and provide the information on how the bitmap bits are organized. The application does this by creating a packed DIB in memory and passing the address of this structure to the DIB driver when it (the application) requests the creation of a DC (Device Context). The packed DIB structure contains a BITMAPINFO structure at its start which gives the width and height of the bitmap and also describes its color table and pixel color organization. Directly following the header are the bitmap bits, organized the same way as the bitmap bits in a DIB file.

What the DIB Driver Can do for You

By using the DIB driver to create a DC in a piece of memory you have allocated and understand the organization of, you can choose to create images in that DC in two ways: use GDI operations in the same way as for any other DC or manipulate the bits directly in memory.

For an animation application this means that you can use the DIB Driver to manage the off-screen image buffer. You create a packed DIB the same size and organization as the window in which the animation will play and then use the DIB Driver to create a DC for it. Thereafter you can do all your image rendering to the off-screen DIB and use the StretchDIBits function to move areas of the off-screen DIB to the display window DC.

Drawing to the DIB Driver DC

As mentioned above, you can perform regular GDI operations on a DIB Driver DC. Since the DIB Driver is a non-palletized device there is no point in selecting/realizing a palette in the DC. There is no harm in doing this however as the driver simply ignores the request. Generic code which selects and realizes a palette when drawing to an arbitrary DC will still work OK if used on a DIB Driver DC.

DIB Blt Functions

There is no function to directly blt a packed DIB to a DC but there are a small number of functions designed to move DIB bits between a DIB and a DDB or the screen DC. Table 1 lists the functions.

Name	Description
SetDIBits	Move DIB bits to a DDB
GetDIBits	Move DDB bits to a DIB
SetDIBitsToDevice	Move DIB bits to a device
StretchDIBits	Move DIB bits to a device and optionally stretch them.

Table 1. DIB Functions

The SetDIBits and GetDIBits functions are directly implemented by screen and printer device drivers. The SetDIBitsToDevice function is a hangover from the Windows 3.0 DIB development and should not be used in new projects. Use StretchDIBits instead of SetDIBitsToDevice. StretchDIBits is the 'do all' function used to transfer DIB images between DCs. It is optionally implemented by the screen device driver. If not implemented in the device driver, GDI uses combinations of SetDIBits, GetDIBits and BitBlt to emulate it. Having GDI emulate StretchDIBits means awful performance. More about performance issues later.

StretchDIBits

Understanding what is going on with the color table entries when StretchDIBits is called can be quite a problem. The following set of figures show what happens when StretchDIBIts is used to draw DIBs to DC's of both palletized and non-

paletized devices. The source DIB color table can be either RGB values (the DIB_RGB_COLORS flag is used) or palette index values (the DIB_PAL_COLORS flag is used). The four possible combinations are explained here.

Figure 4. Using StretchDIBits with DIB_RGB_COLORS to draw on a paletized device DC.

When drawing RGB values to a paletized device, GDI translates each RGB value to an index in the current physical palette by calling an internal version of GetNearestPaletteIndex. The set of indices is then passed to the device driver. The driver tests the index set to see if it is an identity palette. An identity palette is one which has an exact 1:1 mapping of logical palette index values to the current physical palette. In other words the palette table contains the values 0, 1, 2, ... 255. If the driver detects an identity palette then the DIB bits can be moved directly to the screen memory without translation. If the palette is not an identity palette then each pixel value (palette index) in the source DIB has to be translated through the lookup table supplied by GDI to the correct physical palette index value.

µ §

Figure 5. Using StretchDIBits with DIB_PAL_COLORS to draw on a paletized device DC.

When drawing palette index values to a paletized device, GDI builds a translation table to map the logical index values in the DIB to the current physical palette index values. The set of indices and the translation table are then passed to the device driver. If the palette is found to be an identity palette no translation is performed. (See the previous example).

µ §

Figure 6. Using StretchDIBits with DIB_RGB_COLORS to draw on a non-paletized device DC.

When drawing RGB values to a non-paletized device, no translation is done by GDI. The driver gets the RGB values directly. In the case of the DIB driver and an 8 bit DIB, these RGB values are converted to 8 bit pixel values by looking them up in the (DIB Driver) DIB color table.

µ §

Figure 7. Using StretchDIBits with DIB_PAL_COLORS to draw on a non-paletized device DC.

When drawing palette index values to a non-paletized device, GDI looks up each index in the currently selected logical palette and translates it to an RGB value. The table of RGB values is passed to the device driver.

Palettes

Most of the popular display cards today provide 640 by 480 with 256 colors and Windows provides palette management for the available colors. Any application wishing to use these colors will need to create and use one or more palettes. If you are not familiar with how palettes work in Windows, please refer to Ron Gery's articles: "The Palette Manager: How and Why" and "Using DIBs with Palettes" elsewhere on the Developer Network CD.

In designing an animation application, we must consider how color will be used and create one or more palettes accordingly.

Palette Operations Take Time

Each time an application selects a palette for use by itself, Windows creates a mapping between the colors in the requested (logical) palette and the system (physical) palette. Exactly how it does this and the rules governing it are covered in Ron's articles. The important point is that the process of mapping a logical palette to the system palette takes a finite time and is not something we want to be doing every time we draw an image. In fact, GDI attempts to be helpful here and recognizes when the palette being realized in the current foreground application is the same as the one realized before it, and doesn't repeat the matching operation needlessly. None the less, we need to be careful about our use of palettes or the performance of our application will suffer.

One Palette Fits All

For the SPRITES application I chose to use only one palette. Rather than build the palette into the application I chose to always create the palette from the color table contained in the DIB used for the background scene. This was purely a convenience for me. You could just as easily read a palette in from any DIB file and use that one.

If all the images are to look reasonable when rendered with one common palette then the choice of the colors in that palette is obviously very important. How you should go about choosing those colors is beyond the scope of this article. I chose mine the cowards way, by letting the scanning software chose it for me. It so happens that the scanner I used created a common palette to save all of the images I scanned. It's not a great palette but it's OK for the purpose of demonstrating the principle. The scanner used was a Hewlett Packard ScanJet IIc and the software was Hewlett Packard's DeskScan II version 1.5.

If you have several images and want to play with their palettes, try running the

BitEdit and PalEdit tools available in the Video for Windows SDK. If you just want to experiment, try looking at the images in the SPRITES application.

Comment: Add a button to run BitEdit here.

Creating a Palette

Creating a palette from the color table of a DIB is reasonably straight forward once you know how many colors you want to use. A LOGPALETTE structure is created large enough for the number of colors, the color information is copied from the DIB header and a call made to CreatePalette. The LOGPALETTE structure is then freed. Since this is a common requirement when dealing with DIBs, I wrote a function to create a palette directly from the BITMAPINFOHEADER of a DIB. This function is called CreateDIBPalette and is in the palette.c module of the SPRITES sample code. Here it is with the comments and some of the error handling code removed:

```
HPALETTE CreateDIBPalette(LPBITMAPINFO lpBmpInfo)
{
   LPBITMAPINFOHEADER lpBmpInfoHdr;
   HANDLE hPalMem;
   LOGPALETTE *pPal;
   HPALETTE hPal;
   LPRGBQUAD lpRGB;
   int iColors, i;

   lpBmpInfoHdr = (LPBITMAPINFOHEADER) lpBmpInfo;
   if (!IsWinDIB(lpBmpInfoHdr)) return NULL;

   lpRGB = (LPRGBQUAD)((LPSTR)lpBmpInfoHdr + (WORD)lpBmpInfoHdr-
>biSize);
   iColors = NumDIBColorEntries(lpBmpInfo);
   if (!iColors) return NULL;

   hPalMem = LocalAlloc(LMEM_MOVEABLE,
                  sizeof(LOGPALETTE) + iColors * sizeof(PALETTEENTRY));
   if (!hPalMem) return NULL;
   pPal = (LOGPALETTE *) LocalLock(hPalMem);
   pPal->palVersion = 0x300; // Windows 3.0
   pPal->palNumEntries = iColors; // table size
   for (i=0; i<iColors; i++) {
      pPal->palPalEntry[i].peRed = lpRGB[i].rgbRed;
      pPal->palPalEntry[i].peGreen = lpRGB[i].rgbGreen;
```

```
        pPal->palPalEntry[i].peBlue = lpRGB[i].rgbBlue;
        pPal->palPalEntry[i].peFlags = 0;
    }

    hPal = CreatePalette(pPal);
    LocalUnlock(hPalMem);
    LocalFree(hPalMem);

    return hPal;
}
```

The CreateDIBPalette function uses NumDIBColorEntries to get the number of colors in the DIB color table. NumDIBColorEntries copes with the somewhat obscure rules governing exactly how many color entries there are in a DIB color table. The structure contains a **biClrUsed** field which should have the number of colors in it but often is set to zero meaning that the color table is the maximum size appropriate for the color depth.

All of the DIB management code in the examples assumes that the DIB is in Windows format (having been converted from PM format if required when it was loaded). The helper function **IsWinDIB** is used to make the test. More about the DIB support functions later.

The SPRITES Sample Application

I decided to make my sample some form of sprite (cast based) animation for several reasons. Firstly it's much more challenging than frame animation and I'm a sucker for punishment. Secondly, all the techniques required for frame animation are used in sprite animation so an example of sprite animation code also effectively provides a frame example. Of course, the real reason is that I could scan some images to make the sprites which meant less artwork!

An Overview

The SPRITES application uses a DIB for a background scene and allows the loading of multiple sprites on top of the background scene. Each sprite has x,y and z coordinates and optional x and y velocity. It also has a flag to say if it can be dragged by the mouse or not.

A background and set of sprites can be combined into a scene described in a simple INI file. The entire scene can be loaded using the 'Load Scene' item from the 'File' menu.

The application updates the positions of all sprites which have a non-zero velocity as fast as it can using a PeekMessage loop.

Sprites which have the selectable attribute set can be dragged with the mouse. Double clicking on a sprite brings up a dialog which allows all the sprite attributes to be set.

The Z order value of zero is the front most position and values greater than zero go towards the back. I used a maximum value of 100 but the limit is actually 65535.

A separate debug information window works in conjunction with 'dprintf' statements in the code and the 'Debug' menu to show what's going on.

Figure 8 shows how the background DIB, the DIB Driver DC, common palette and screen DC relate to each other.

µ §µ §
Figure 8. The architecture of the SPRITES application.

Code Notes

The SPRITES sample code has some features which I will mention here to avoid confusion when you read the code.

There are several 'dprintf' statements throughout the code. These are used to print debugging information in the debug window. The amount of information is controlled by the current debug level which can be set from the 'Debug' menu. I try to use level 1 for error messages (most important), level 2 for general procedural steps (like entering a major function), level 3 for increased procedural detail (what's happening in the function) and level 4 for data dumps and things that go on too fast to want to have data about them all the time. The dprintf statements are implemented as macros in GLOBAL.H.

Since the world of Windows programming is moving rapidly towards 32 bits, I try not to include the near or far attribute in pointer names. So instead of npDIB, lpDIB or fpDIB I simply use pDIB. This might seem confusing since it is so common to see lpSomething in Windows code but I believe that it will help in the long run. Almost all pointers in the code are actually FAR pointers. There are also a few HUGE pointers for dealing with big objects and one or two near pointers to data in the local heap where this made a significant difference to the performance.

In many cases when code fragments are included in the text of the article, I have removed comments, debug code and sometimes error reporting statements to help

clarify what I'm talking about. Please look at the actual code in the sample before writing your own.

Be aware that last minute changes to the code before publication might mean some slight differences from what's in the article and what's in the sample code. If in doubt, go with the code in the sample.

The Background

The background scene of the animation is a single DIB. The color table found in the background DIB is used to create the palette used to render all the images to the window DC. A background is loaded by using the 'Load Background' item from the 'File' menu or by loading a scene by using the 'Load Scene' item. The LoadBackground function in BKGND.C is responsible for doing the work. We'll look at the code for each step of the function with a brief description of what is happening at each stage.

```
DeleteSpriteList();
```

The current set of sprites is deleted. This isn't really necessary, but it greatly simplified the set of dependencies. The background provides the common palette and each sprite has its color table adjusted to fit the background color set so it was just easier to start again each time a background was loaded.

```
DeleteDIB(pdibBkGnd);
```

The existing background DIB is deleted. The DeleteDIB function does nothing if the DIB does not currently exist.

```
pdibBkGnd = LoadDIB(pszPath);
```

The new background DIB is loaded. If no path was provided for the background DIB, a dialog is presented to choose it.

```
if (hpalCurrent) DeleteObject(hpalCurrent);
hpalCurrent = CreateDIBPalette(pdibBkGnd);
```

Any current palette (from a previous background DIB) is deleted and a new one created from the new background DIB. The CreateDIBPalette function is in the DIB.C module.

It is at this point that the palette could be modified to ensure that the first 10 and last 10 entries exactly match the system color entries. This is important to do so

that the palette indicies will not need translation when they are blted to the screen. (This is discussed more elsewhere).

The window rectangle is then adjusted to fit the new background. I'll skip the code for that since it's commonplace.

```
   if (hdcOffScreen) {
      DeleteDC(hdcOffScreen);
      hdcOffScreen = NULL;
   }
   DeleteDIB(pdibOffScreen);
```

Any existing off-screen DC and its associated DIB are deleted. The DeleteDIB function is in DIB.C.

```
   pdibOffScreen = CreateCompatibleDIB(pdibBkGnd);
```

A new off-screen DIB is created the same size as the background DIB. The CreateCompatibleDIB function is in DIB.C.

```
   hdcOffScreen = CreateDC("DIB", NULL, NULL, (LPSTR)pdibOffScreen);
```

A new off-screen DC is created using the DIB Driver and the new off-screen DIB. Note that the DIB Driver requires that the last argument be a pointer to a packed DIB structure.

```
   if (!pPalClrTable) {
      pPalClrTable = (LPBITMAPINFO)
ALLOCATE(sizeof(BITMAPINFOHEADER)
                                 + 256 * sizeof(WORD));
   }

   _fmemcpy(pPalClrTable,
         pdibOffScreen,
         sizeof(BITMAPINFOHEADER));

   pIndex = (LPWORD)((LPSTR)pPalClrTable +
sizeof(BITMAPINFOHEADER));
   for (i=0; i<256; i++) {
      *pIndex++ = (WORD) i;
   }
```

A 1:1 color lookup table is required when StretchDIBits is used later to copy image data from the off-screen DC to the Window DC. If the table doesn't already exist, the memory is allocated for it. The table header is copied from the background DIB. This sets the size information to be the same as the background and off-screen DIBs. Lastly, the color table is filled with the values 0 through 255 which gives the 1:1 color index mapping we will need later.

```
   Redraw(NULL, bUpdateScreen);
```

A call is made to render the background image to the off-screen DC and to update the window DC with the new image.

The Sprites

Each sprite consists of a DIB which provides the image and a set of variables which describe its size, position and optionally its velocity. The information about each sprite is contained in a SPRITE structure:

```
typedef struct _SPRITE {
   struct _SPRITE FAR *pNext;  // pointer to the next item
   struct _SPRITE FAR *pPrev;  // pointer to the prev item
   PDIB pDIB;                  // The DIB image of the sprite
   int x;                  // X Coordinate of top-left corner
   int y;                  // Y Coordinate of top-left corner
   int z;                  // Z order for sprite
   int vx;                  // X velocity
   int vy;                  // Y velocity
   int width;                // width of bounding rectangle
   int height;               // height of bounding rectangle
   BYTE bTopLeft;            // top left pixel value
   COLORREF rgbTopLeft;      // top left pixel color
   BOOL bSelectable;         // TRUE if sprite can be mouse selected
} SPRITE, FAR *PSPRITE;
```

The transparent regions of the sprite are determined by the color of the top-left pixel of its DIB. When the DIB is authored one color is reserved (it doesn't matter what color it is) for the transparent regions and they are all filled with that color. The top-left pixel is also set to that color. This is easy to achieve in practice since it is natural that the corners of the sprite rectangle are transparent for most real object shapes.

To see how these parameters are set, let's look at the LoadSprite function in

```
pSprite = (PSPRITE) ALLOCATE(sizeof(SPRITE));
```

Memory is allocated for the SPRITE structure.

```
pSprite->pDIB = LoadDIB(pszPath);
```

The DIB image of the sprite is loaded. If no path was supplied to LoadSprite a dialog is presented to select the DIB.

```
pSprite->width = (int) DIB_WIDTH(pSprite->pDIB);
pSprite->height = (int) DIB_HEIGHT(pSprite->pDIB);
pSprite->x = 0;
pSprite->y = 0;
pSprite->z = 0;
pSprite->vx = 0;
pSprite->vy = 0;
pSprite->bSelectable = TRUE;
pSprite->pNext = NULL;
pSprite->pPrev = NULL;
```

The defaults are set for the sprite parameters.

```
MapDIBColorTable(pSprite->pDIB, pdibBkGnd);
```

The color table in the DIB is mapped to the color table of the background DIB. This isn't required if all the sprite DIBs are authored with the same color table as the background DIB. If the color tables differ, this at least renders the sprite image in the best way possible rather than as a collection of seemingly random colors. The MapDIBColorTable function is in DIB.C.

```
pSprite->bTopLeft = GetDIBPixelValue(pSprite->pDIB, 0, 0);
pSprite->rgbTopLeft = GetDIBPixelColor(pSprite->pDIB, 0, 0);
```

The index value and color of the top-left pixel are saved for later use in determining the transparent areas of the image.

```
if (pSpriteList) {
   pSpriteList->pPrev = pSprite;
   pSprite->pNext = pSpriteList;
}
pSpriteList = pSprite;
```

The sprite is added to the top of the sprite list. For now the position in the list doesn't matter. It will be adjusted when the Z order is set.

```
SetSpriteZOrder(pSprite, 50, NO_UPDATE);
```

The Z order of the sprite is set to a default value. Z order zero is the front most sprite. Sprites with the same Z order are drawn with the one at the top of the sprite list front most. Painting is done from the bottom of the list to the top. The list is always maintained in Z order so don't set the Z order value directly, use the SetSpriteZOrder function which correctly manages the list.

```
if (bRedraw != NO_UPDATE) {
   GetSpriteRect(pSprite, &rc);
   Redraw(&rc, UPDATE_SCREEN);
}
```

If redrawing the sprite was requested, the sprite rectangle is added to the redraw list. For a single sprite loaded manually from the menu it needs to be redrawn to be visible, but if the sprite is being loaded along with other sprites to form a scene then the redraw operation only needs to be done when all the sprites are loaded.

The DIB Functions and Macros

Since Windows provides no functions to deal with DIBs the way it does with bitmaps (DDBs) we need to create our own. The DIB.C module contains all the DIB handling functions used in the application except those used for rendering which are in the DRAW.C module. A brief description of each function is given here. See the code for more details. A set of macros are defined in GLOBAL.H for accessing various parameters of a DIB. They are also described briefly here.

The DIB Macros

Macro name	**Function**
DIB_WIDTH(pDIB)	Image width
DIB_HEIGHT(pDIB)	Image height

DIB_PLANES(pDIB)	Number of color planes
DIB_BITCOUNT(pDIB)	Number of bits per pixel
DIB_CLRUSED(pDIB)	Number of colors used
DIB_COLORS(pDIB)	Number of colors
DIB_PCLRTAB(pDIB)	Pointer to the color table
DIB_BISIZE(pDIB)	Size of the BITMAPINFO struct
DIB_PBITS(pDIB)	Pointer to the bits
DIB_PBI(pDIB)	Pointer to the BITMAPINFO struct
DIB_STORAGEWIDTH(pDIB)	Scan line storage width

Table 2. DIB Macros

The DIB Functions

PDIB LoadDIB(LPSTR pszPath)

Load a DIB. pszPath is the file path or NULL to invoke the file open dialog.

void DeleteDIB(PDIB pDIB)
Delete a DIB. If pDIB is NULL, the request is ignored.

BYTE GetDIBPixelValue(PDIB pDIB, int x, int y)
Get the value (color table index) of a pixel at coordinates x,y of the DIB pointed to by pDIB.

COLORREF GetDIBPixelColor(PDIB pDIB, int x, int y)
Get the color (RGB) of a pixel at coordinates x,y of the DIB pointed to by pDIB.

BOOL IsWinDIB(LPBITMAPINFOHEADER pBI)
Test if a DIB is Windows format (rather than Presentation Manager format).

void ShowInfo(LPBITMAPINFO lpBmpInfo)
A debugging function to display attributes of a DIB in the debug window.

WORD NumDIBColorEntries(LPBITMAPINFO lpBmpInfo)
Get the number of colors in the color table of a DIB. Used in the DIB_COLORS macro.

PDIB CreateCompatibleDIB(PDIB pOld)
Create a new DIB the same size and color format as an existing DIB.

HPSTR GetDIBPixelAddress(PDIB pDIB, int x, int y)
Get a pointer to the pixel at address x,y in the DIB pointed to by pDIB.

void MapDIBColorTable(PDIB pdibObj, PDIB pdibRef)
Map the colors in the color table of the DIB pointed to by pdibObj to the colors in the color table of the DIB pointed to by pdibRef. This is done by creating a temporary DIB Driver DC the same size as the object DIB, with the color table of the reference DIB. The object DIB is then rendered to the DIB Driver DC using StretchDIBBits with the DIB_RGB_COLORS option. The resulting bits (now mapper tot he reference color table) are copied back to the object DIB bits.

The Drawing Functions

The module DRAW.C contains all of the functions to render images to the off-screen DIB Driver DC and to the window DC. The most important functions are RenderDIBBitsOffScreen, Redraw and Paint.

RenderDIBBitsOffScreen

This function is used to render the background DIB and the sprites to the off-screen DC. It uses two functions: CopyDIBBits and TransCopyDIBBits in the FAST32.ASM module to perform the actual bit transfers. Here's a description of the function:

```
rcDraw.top = rcDraw.left = 0;
rcDraw.right = DIB_WIDTH(pdibOffScreen) - 1;
rcDraw.bottom = DIB_HEIGHT(pdibOffScreen) - 1;

if (prcClip) {
   if (!IntersectRect(&rcDraw, &rcDraw, prcClip)) return;
}

rcDIB.left = x;
rcDIB.right = x + DIB_WIDTH(pDIB) - 1;
rcDIB.top = y;
rcDIB.bottom = y + DIB_HEIGHT(pDIB) - 1;

if (!IntersectRect(&rcDraw, &rcDraw, &rcDIB)) return;
```

The function is supplied with a clipping rectangle describing the area to be drawn into. The first step is to intersect that rectangle with the off-screen DIB boundary so we don't try to draw off the DIB. If there is no intersection then there is nothing to do. The resultant rectangle is intersected again, this time with the bounding rectangle of the DIB itself to ensure we aren't going to be doing more work than is really necessary. Again, if there is no intersection, the DIB isn't visible and the function returns.

```
pStartS = GetDIBPixelAddress(pDIB,
                  rcDraw.left - x,
                  rcDraw.bottom - y);

pStartD = GetDIBPixelAddress(pdibOffScreen,
                  rcDraw.left,
                  rcDraw.bottom);

lScanS = DIB_STORAGEWIDTH(pDIB);
lScanD = DIB_STORAGEWIDTH(pdibOffScreen);
```

The address of the bottom-left corner of the draw rectangle is found in both the source DIB and the destination DIB. The length of the physical scan line for each DIB is obtained. The addresses represent the lowest address of the DIB bits we need to copy.

```
if (!bTrans) {
   CopyDIBBits(pStartD,
         pStartS,
```

```
            rcDraw.right - rcDraw.left + 1,
            rcDraw.bottom - rcDraw.top + 1,
            lScanD,
            lScanS);
   } else {
      TransCopyDIBBits(pStartD,
               pStartS,
               rcDraw.right - rcDraw.left + 1,
               rcDraw.bottom - rcDraw.top + 1,
               lScanD,
               lScanS,
               bTranClr);
   }
```

If the DIB is to be treated as non-transparent (as for the background DIB) then the CopyDIBBits function is called. If the DIB has a transparency color associated with it (as for a sprite DIB) then the TransCopyDIBBits function is used. These two copy functions are implemented in 32 bit assembler in FAST32.ASM.

Redraw

This function is used in two ways depending on whether the screen needs to be updated or not. When rendering multiple images, it is important to do the least amount of work so the function is used to add items to the redraw list and optionally to do the actual redraw. Here's what goes on:

```
   if (prcClip) {
      AddDrawRectItem(&DrawList, prcClip);
   } else {
      if (pdibBkGnd) {
         rcAll.left = rcAll.top = 0;
         rcAll.right = DIB_WIDTH(pdibBkGnd);
         rcAll.bottom = DIB_HEIGHT(pdibBkGnd);
         AddDrawRectItem(&DrawList, &rcAll);
      }

   }
   if (bUpdate == NO_UPDATE) return;
```

The first stage is to add the supplied clipping rectangle to the redraw list. If NULL was specified, this is interpreted as meaning that the entire window needs to be redrawn and this causes a rectangle the size of the background DIB to be added to the list instead. If no request was made to update the screen, the function exits here.

```
MergeDrawRectList(&DrawList);
pLastSprite = pSpriteList;
if (pLastSprite) {
   while (pLastSprite->pNext) pLastSprite = pLastSprite->pNext;
}
if (bUpdate == UPDATE_SCREEN) {
   hDC = GetDC(hwndMain);
}
pDrawRect = DrawList.pHead;
while (pDrawRect) {
   RenderDIBBitsOffScreen(pdibBkGnd,
                          0, 0,
                          &(pDrawRect->rc),
                          0,
                          FALSE);
   pSprite = pLastSprite;
   while (pSprite) {
      RenderSpriteOffScreen(pSprite, &(pDrawRect->rc));
      pSprite = pSprite->pPrev;
   }
   if (bUpdate == UPDATE_SCREEN) {
      Paint(hDC, &(pDrawRect->rc));
   }
   pDrawRect = pDrawRect->pNext;
}
if (bUpdate == UPDATE_SCREEN) {
   ReleaseDC(hwndMain, hDC);
}
EmptyDrawRectList(&DrawList);
```

The first step in the rendering process is to merge all the overlapping rectangles in the redraw list. This results in a list of non-overlapping rectangles and gives us the least area which will need to be modified. Pointers are obtained to the last sprite in the sprite list and the first rectangle in the redraw list. The sprite list is walked form bottom to top so that high Z order sprites (which are at the bottom of the list) appear at the back of the scene.

Then for each rectangle, the background is replaced in the off-screen DC by calling RenderDIBBitsOffScreen. The sprite list is then walked, rendering each sprite. The clipping of each sprite is handled in part by the RenderSpriteOffScreen function and in part by the RenderDIBBitsOffScreen function.

If Redraw was called with the UPDATE_SCREEN flag, the screen DC is repainted by calling the Paint function for the current draw rectangle.

When all of the rectangles in the list have been redrawn, the list is reset to empty by a call to EmptyDrawRectList.

All of the drawing rectangle functions can be found in DRAW.C.

Paint

The Paint function handles updating the screen DC from the off-screen DIB Driver DC. Here's the code:

```
if (prcClip) {
    w = prcClip->right - prcClip->left;
    h = prcClip->bottom - prcClip->top;
    xs = xd = prcClip->left;
    yd = prcClip->top;
    ys = DIB_HEIGHT(pdibOffScreen) - prcClip->bottom;
} else {

    w = DIB_WIDTH(pdibOffScreen);
    h = DIB_HEIGHT(pdibOffScreen);
    xs = xd = ys = yd = 0;
}
```

The width and height of the rectangle to be copied and the start point in the off-screen DC and the window DC are computed based on either the supplied clipping rectangle or the size of the off-screen image.

```
if (hpalCurrent) {
    hOldPal = SelectPalette(hDC, hpalCurrent, 0);
    RealizePalette(hDC);
}
```

The current palette (obtained originally from the background DIB color table) is selected into the screen DC and realized. This operation only takes any significant amount of time the first time it is called after the application has become the foreground application.

```
StretchDIBits(hDC,          // dest dc
        xd,             // dest x
        yd,             // dest y
```

```
        w,              // dest width
        h,              // dest height
        xs,             // src x
        ys,             // src y
        w,              // src width
        h,              // src height
        DIB_PBITS(pdibOffScreen), // bits
        pPalClrTable,   // BITMAPINFO
        DIB_PAL_COLORS, // options
        SRCCOPY);       // rop
```

StretchDIBits is called to copy the bits of the off-screen image to the screen memory. Note the use of DIB_PAL_COLORS and the 1:1 color lookup table (pPalClrTable). See the section on performance below for more details.

```
    if (hOldPal) SelectPalette(hDC, hOldPal, 0);
```

The palette in the screen DC is restored.

Updating Positions

The UpdatePositions function really has very little to do with creating sprites. I included this to show off the performance by whizzing a few images across the screen. The code walks down the sprite list looking for any sprites which have a non-zero velocity vector. When it finds one it adds the current position to the redraw list so that it will be erased from there. The sprite position is updated and the new position of the sprite added to the redraw list. In both cases the Redraw function is called with the NO_UPDATE option to prevent any actual draw operations until we are done walking the list.

Once all the sprites have been updated a test is made to see if any changes occurred and if so the Redraw function is called one more time, but this time with the UPDATE_SCREEN option which causes the redraw rectangle list to be walked and redrawn with all the updates being reflected to the screen DC.

If you want to do special case sprite movements (funny trajectories, flipping sprite images as they move and so on) this is the place to add the code.

The Scene File Format

To make life easier while debugging the SPRITES application, I created an INI file to describe what background DIB to use and what sprites to load. The format

isn't very tidy but it's documented here so that you can create your own files. A nice improvement to the application would be an option to save the current scene in a file.

Here's part of the GARDEN.INI file showing the description of the background and two of the sprites:

```
[Background]
dib=bkgnd.dib
```

This describes the background DIB to use. There is only one entry to specify the name of the DIB file. The default is to load the file from the current directory. If this isn't what you want, put in the full path.

```
[Sprites]
sun=1
cloud2=1
```

This section gives a list of sprites to be loaded. The '=1' bit is bogus. Only the list of names are used by the loader code.

```
[Sun]
x=400
y=0
z=99
dib=sun.dib
selectable=0
vy=1
```

This describes the 'sun' sprite listed in the [sprites] section.

```
[Cloud1]
x=150
y=20
z=80
vx=2
dib=cloud16.dib
```

This describes one of the 'cloud sprites listed in the [sprites] section.

Table 3 shows the list of attributes for a sprite, a description of each one and the default if the entry is omitted.

Key	Description	Default value
dib	The path of the DIB file.	(none) This is a required entry.
x	The X position of the top-left corner of the sprite.	0
y	The Y position of the top-left corner of the sprite.	0
z	The Z order value. O is front most. More than one sprite can have the same Z order value.	50
vx	The X velocity in pixels per redraw cycle.	0
vy	The Y velocity in pixels per redraw cycle.	0
selectable	If the sprite can be selected for dragging by the mouse.	1 (DIB can be selected)

Table 3. The sprite attributes.

Note that you cannot enter negative values directly because the code uses GetPrivateProfileInt to retrieve them. Negative values must be entered as 16 bit two's compliment values. Enter -1 for example) as 0xFFFF or 65535.

Note: This is an example of a stupidly named Windows function. It says 'Int' in the name but always works with unsigned ints. This rather frustrating and brain-dead feature will unfortunately be with us for life. Such is the mission of backward compatibility in Windows.

Measuring Performance

One of the problems with this sort of work is that analysis of the results can be rather subjective which is something I detest. Often I hear: "Hmm, I'm sure it was faster before you did that". It's very nice to be able to respond with a list of timings and prove the point objectively. To this end I have used two different ways to measure how long various operations in the code took to execute.

Using Software Timing

The first timing technique involves reading the system clock at the start and end of a section of code and reporting the elapsed time. This technique is OK with two provisos: that the time measurements are accurate enough and that displaying the timing information does not contribute significantly to the times being measured.

Timing measurements were done with the MMSYSTEM function timeGetTime rather than the regular Windows function GetTickCount because timeGetTime returns a millisecond count accurate to the nearest millisecond and GetTickCount returns a millisecond count only accurate to the nearest 55 ms (one DOS clock tick - hence the name).

Displaying the results using the debug print statements (dprintf) is however rather invasive and if you use this technique you must be aware that painting the debug information window is a slow process and while this might not contribute to the execution time of the piece of code you are measuring, it certainly does contribute to the overall execution time of the application and hence slows down the animation significantly. Using a small (two or three lines) debug window helps to minimize this.

Despite the invasive nature of showing the results, the timings of small sections of code are quite accurate and very helpful in determining whether optimizing a particular piece of code is worthwhile and when the work has been done, what the improvement was.

Using Hardware Techniques

The second technique I used is covered in another article: "Use Your Printer Port to Measure System Timings" by the same author. This technique sets and clears bits of a printer port at various places in the code. By using an oscilloscope to monitor when the transitions of the bits take place, accurate timing measurements can be made. Please see the article mentioned above for details of how this technique works.

Using the oscilloscope to monitor various operations in the code I discovered several performance problems, the most notable being that allocating small blocks of memory with GlobalAlloc is very slow and inefficient.

Improving Performance

While creating the SPRITES sample I measured the execution times of various bits of my code in order to determine where the bottlenecks were. This section describes what I found out.

Memory Allocation

The first discovery I made when I started looking at performance was that using GlobalAlloc to allocate all of the dynamic memory blocks in the application isn't the best choice. I had defined a macro to allocate and free memory blocks so that the technique could be easily changed if required. I had decided to allocate all memory the same way - which essentially ruled out using local memory for anything. This was a big mistake. I was using my ALLOCATE macro (which calls GlobalAlloc) to allocate the elements of the redraw rectangle list. Using the oscilloscope, I measured the time taken to be almost 3 ms on my 386/33 Compaq (in Enhanced Mode). I was quite disgusted by this and rewrote the rectangle functions to use local memory blocks allocated with LocalAlloc and near pointers to access the structures. The result was that the execution time of AddDrawRectItem fell from about 3 ms to about 50 us. This cut almost 6 ms of the 72 ms cycle time of one of my test scenes which has a single sprite (a dog) moving one pixel at a time from left to right across a plain blue background.

The memory for the SPRITE structure could also be allocated locally and the

FAR pointers used within the structure for pPrev and pNext replaced with near pointers. This might give a small improvement in the rendering time since the sprite list is walked quite often.

Rendering DIBs to the Off-Screen DIB

The first try at rendering the sprite DIBs to the off-screen DC was a complete disaster. I tried to use raster ops to create monochrome masks for the transparency regions of each sprite and to insert the non-transparent areas into the off-screen DIB. After many hours of fruitless work and a phone call or two I was reminded that the DIB driver supports a background mode called NEW_TRANSPARENT and if this is set, StretchDIBits will treat the current background color as the transparency color of a bitmap. The resulting code is implemented in the RenderDIBOffScreen function in DRAW.C.

Close examination of the code shows that the call to StretchDIBits is made using the DIB_RGB_COLORS option which causes the RGB values in the color table of the source DIB to be mapped to the RGB values of the color table in the destination DIB (the off-screen DIB in this case). This is vary handy if your two DIBs have different color tables but is rather a waste of time if they are the same. Even assuming that you wanted to use DIBs with different color tables, it seems ridiculous to do the mapping every time the DIB is rendered. Why not do the mapping once and save the results.

To do this I created the MapDIBColorTable function in DIB.C. This function creates a temporary DC using the DIB driver, renders the DIB to the DC using StretchDIBits with the DIB_RGB_COLORS option and then copies the resulting bits in the DIB driver DC DIB back to the original source DIB. Net result is that the pixels in the source DIB are now mapped to the color table supplied by the reference DIB (the background DIB in our case).

So far, so good. I then got way too clever and modified the StretchDIBits call in the RenderDIBOffScreen function to use DIB_PAL_COLORS with a 1:1 color table (as is used in the Paint routine). This didn't work - I got weird colors. After some research I discovered that you can't use DIB_PAL_COLORS when working with a non-palletized device and the DIB Driver is a non-palletized device. What happens is that GDI tries to be helpful. It detects that the device driver isn't palletized, maps the palette indices to RGB values and sends those to the driver. Net result - a waste of processor time, and the wrong colors. There is a way around this problem however. In MMSYSTEM.H there is a macro called DIBINDEX which can be used to define the colors in the DIB color table as palette index values:

```
clr[n] = DIBINDEX[n]
```

This sets a special flag in the COLORREF structure which the DIB driver recognizes and instead of matching the RGB values normally found in the color table it treats the low byte as a palette index. But by this time I'd lost interest in using StretchDIBits and decided to do my own thing.

I decided to implement a simple function to simply copy the bits of the sprite DIB to the off-screen DIB (ignoring the transparency problem for the moment). This worked great. The performance was way better and the colors were right. I added some code to implement the transparency feature (all of this in C) and although I now had transparent sprites again, the performance went way back down. The cause of the loss of performance being the code generated by the C compiler which dealt with all the huge pointers I was using. Still the idea seemed good.

I recoded the function to copy a scan line of the DIB in assembler and found that the speed improved quite a lot. I improved the assembler function to copy a whole block rather than calling it multiple times to copy scan lines and the result (which is in FAST16.ASM) was another small speed improvement.

The final touch was to use a technique developed by Todd Laney to create a 32 bit code segment for the assembler routine to run in. This code can be found in FAST32.ASM and the macros which support it in CMACRO32.INC. The code was taken from FAST16.ASM and optimized by Todd. There are several rules regarding using 32 bit code segments and how to generate them. These are detailed in the section: "Using 32 Bit Code Segments" below.

Table 4 shows some timings measured at various points through the development cycle of this piece of code. The cycle time shown is the time the application takes to complete one move, render, redraw cycle. The background time is the time taken to render the background DIB (which is not transparent) into the area the sprite is being moved from, and the sprite time is the time taken to render the sprite image in the new position. All times are in ms. The sprite was smolivia.dib which is a 256 color image. For all cases the paint time was constant at about 7 ms.

Description	**Cycle**	**Bkgnd**	**Sprite**
StretchDIBits with	3	1	2

DIB_RGB_COLORS	00	2	80
StretchDIBits with DIB_PAL_COLORS	75	35	35
Primitive C code rectangle copy	12	-	-
C code with transparency using huge ptrs	40	1.8	31
Using 16 bit assembler to copy lines	17	2	6
Using 16 bit assembler to copy blocks	16	1.6	5.5
Using 32 bit assembler to copy blocks	16	1.6	4.6
Optimized 32 bit assembler	1	0	2

1	.	.
	6	6

Table 2. Code timings

There is one further improvement which is as yet untried. I expect that using run length encoded DIBs (RLE) to define a sprite might be more efficient. The RLE code scheme allows for blocks of the DIB to be skipped over using a form of relative addressing. This scheme could be used to encode only the visible areas of a sprite. For images which have a lot of transparency, this could give great improvements in the rendering time.

Updating the Window DC from the Off-Screen DIB

The first and most important thing to know here is that you can't use BitBlt to move bits from the DIB Driver DC to the screen DC. BitBlt is a device driver implemented function and each driver only understands its own DC's. The DIB driver owns the DIB Driver (off-screen) DC and the screen device driver owns the window DC. This rule also prevents you from using BitBlt to copy data from a screen DC to a printer DC. Where I most often get caught out by this is in creating compatible DCs. It's very easy to forget that a particular DC was created (and therefore is owned by) a particular device driver. So if you create a DC compatible with an existing printer DC or DIB Driver DC, you still can't use BitBlt to move data between that DC and any DC owned by the screen device driver. This is where the DIB specific functions come in.

Updating the window from the off-screen DIB is reasonably straightforward:

1) The palette (created from the background DIB when it was loaded) is selected into the window DC and realized.

2) StretchDIBits is called with the DIB_PAL_COLORS option and a 1:1 color table.

3) The original palette is reselected into the DC.

That's all it takes. Using the DIB_PAL_COLORS option prevents any color matching. Actually, the first time the palette is realized, GDI establishes a new system palette and modifies the index values in the logical palette so that they map directly to the system palette. Once this has been done there is no need to map a logical palette index to a physical palette index and consequently the process runs very quickly. This mapping stays valid so long as the system palette

doesn't change.

Redraw Rectangle List

Since rendering images to the off-screen DC and copying the changed areas to the main window DC are really what take all the time, an attempt is made to reduce the amount of blitting done to a minimum. To achieve this a list of rectangles is maintained which need to be redrawn on the off-screen DIB and repainted to the window DC. As the first step in performing a redraw, the redraw list is merged so that any overlapping rectangles are combined into a single rectangle. Then, each rectangle in the list is redrawn. In this way the minimum area is modified each time. Use the "ShowUpdateRects" menu item in the "Debug" menu to look at the redraw areas (in cyan) and the repaint to the screen areas (bordered in magenta).

Using Assembler Code

Writing critical sections of the code in assembler can provide two advantages. First, the code is often smaller and faster than that generated by the compiler and second, because of the way Windows allocates descriptor table entries for large (> 64k) memory blocks, you can access an entire block of memory (a packed DIB for example) with the selector to the base of the block and a 32 bit offset. Simply using a 32 bit register (ESI for example) for the offset forces the assembler to include an addressing modifier into the code which results in a 32 bit offset being used. This is how the FAST16.ASM module works.

Using 32 Bit Code Segments

You can include also 32 bit code segments in your application. This is almost exactly the same as the 16 bit assembler case except that the code is now running with 32 bit offsets by default so no addressing modifier needs to be included before an instruction which uses a 32 bit offset. On a 386 there is almost no improvement in performance from this because the execution time of the modifier is so low, but on a 486 the presemce of a modifier alters the instruction pipelining resulting in a very noticable reduction in performance. So using 32 bit code segments gives a minor improvement on a 386 (a few percent) and usually a one and a half to two times improvemetn on a 486. **Using 32 bit code segments has nothing to do with a flat model address space, you are still working with segment and offset.** Currently, the only way to implement this is in assembly language using a special version of the C macros. There are a some regulations governing how these segments must be created and used:

1) The segment must be separate from any 16 bit code segments. To ensure this, name it with a unique name (such as TEXT32) and link it with the

/NOPACKCODE option in the linker to prevent the segment being merged with other segments. Failure to keep the segment isolated will result in a GP fault when one of your 32 bit functions attempts to return to the 16 bit calling code.

2) The 32 bit code segment should not call any 16 bit code. This limits the 32 bit code to doing data manipulation tasks. This is partly to do with the fact that the 32 bit code is running with a 16 bit stack.

3) The assembly code must include the special CMACRO32.INC file. This is a modified version of CMACROS.INC which handles 32 bit code segments.

4) The code must be assembled with MASM 5.1 not MASM 6.0 which is not backward compatible with the macros. A version of the macros compatible with MASM 6.1 will be included with this technote and its sample if they become available.

The way that the 32 bit segments are integrated is quite ingenious. Each procedure has a short piece of code at its start which determines if the segment is running in 32 bit mode. If it is found to be in 16 bit mode, a jump is made to a piece of fix-up code at the start of the segment which modifies the descriptor table entry for the segment to change the mode to 32 bit and then returns to the called function which now executes in 32 bit mode. This fixup only needs to occur once for the segment. Once it has been set to 32 bit mode it stays that way.

There is one minor problem with using 32 bit code segments. The linker sets a bit in the EXE file header in the segment table to mark the 32 bit code segment as being a 32 bit segment - much as you might expect. Windows ignores many of the flags in the segment information including this one. However the ROM Windows version of Windows makes use of this bit to mark a segment as being in ROM which causes disastrous results if you try to run an application with 32 bit code segments in RAM. This problem can be cured by modifying the flag bits in the segment information table of the EXE file. There is currently no tool available to do this.

Sprite Design

Since the performance is limited largely by the amount of data being moved about, make sure that sprites don't have borders around them. Even though the border is transparent, these areas cause trouble because the entire sprite rectangle needs to be redrawn whenever the sprite is moved.

µ §

Figure 9. Good and Bad Sprite Designs

Run Time Decisions

Not all device drivers have the StretchDIBits function implemented. Even in some of the drivers which do have it implemented, the implementation may not be all that good. If StretchDIBits is not implemented in the driver then GDI will simulate it by using combinations of SetDIBits and BitBlt. This requires the use of an intermediate DDB which GDI uses to band the image in 64k chunks. This is horrendously slow. The Video for Windows code tests the perfomrance of StretchDIBits when it starts up by timing a few calls to StretchDIBits against calls to SetDIBits and BitBlt for the same image. If StretchDIBits is slower, then the driver is either missing the StretchDIBits function or has a very bad implementation. In any case, the Video for Windows code reports the problem to the user and carries on using the SetDIbits and BitBlt option. Note that although the application still runs without a device driver implementation of StretchDIBits it is much slower than it would be if the function were present.

Notes About BitBlt

During the early stages of developing the SPRITES application I tried creating DDBs from the sprite DIBs and then creating monochrome bitmap masks for their transparent areas. The bitmaps were combined together with the background bitmap using various raster operations. Even though BitBlt isn't used anywhere in the final SPRITES code, I though my notes might still be of some interest:

1) When a DC is first created (for example by calling CreateCompatibleDC) it has a default 1 x 1 monochrome bitmap selected into it. So if you subsequently call CreateCompatibleBitmap you will get a monochrome bitmap. This isn't what you'd expect if you started with a color DC, created a compatible DC from that and then asked for a compatible bitmap. To ensure that your bitmap is the same color format as the original DC get the color information by calling GetDeviceCaps for BITSPIXEL and PLANES and then use that information in a call to CreateBitmap:

```
BitsPixel = GetDeviceCaps(hDC, BITSPIXEL);
Planes = GetDeviceCaps(hDC, PLANES);
hbm = CreateBitmap(width, height, Planes, BitsPixel, NULL);
```

2) When using BitBlt to convert a color bitmap to a monochrome bitmap, if the pixel color of the source bitmap is the same as the current background

color then the monochrome output is white (1) otherwise it is black (0).

3) If you want to be able to use blt operations like XOR and AND to do transparency masking onto a DIB dc, the sprite image and mask must also be in DIB DCs. This is because BitBlt cannot work with device contexts from two different drivers.

Unimplemented Features

The application only supports 8 bit DIBs. It won't let you load a 2 or 4 bit DIB. It also doesn't support RLE DIBs.

It turned out that once the code was complete, the DIB Driver had taken rather a back seat in the design and in fact could be removed all together. The only function it provides is to map RGB color values to index values and this could just as easily be done with a small piece of code.

The background DIB is treated as a special case throughout the code. It might have been better to treat it as just another sprite with a Z order value of 65535. When loaded (as the background DIB) it would still be used to create the common palette but once that was done it would simply be added to the tail end of the sprite list.

Some sprites are simply rectangles and in these cases it would be nice to be able to set a flag to say that transparency does not apply to this sprite. This could easily be done by changing the BYTE value used in the bTransColor field of SPRITE to be a UINT (or WORD) with a special reserved vlaue of 0xFFFF meaning that there is no transparent color.

Another possible improvement to the architecture might be to use two off-screen DIBs in a double buffering arangement. Each time an image was rendered to a buffer, the difference between the new image and the old one in the other buffer would be recorded (in a third DIB) and then converted to RLE before being sent to the screen DC. This might look like a lot of work but has the advantage that it minimizes the amount of data going to the screen DC which is a limiting factor. Doing the render, compare and update to the off-screen DIBs could be done very efficiently with a small piece of assembly code.

Windows NT and Chicago Issues

The DIB Driver is not provided in either Windows NT or Chicago. Instead there will be a new API called CreateDIBSection which will allow you to do the same

things but in a slightly different way. If you want to be able to port your application trivially to either Chicago or Windows NT then you should remove the DIB Driver and use your own code to do the RGB to index color conversions and also remove the 32 bit assembler code since this is very platform specific.

The ALLOCATE and FREE macros used for memory management in the sample don't map very well to Windws NT functions. A better approach would be to include the windowsx.h header file and use the GlobalAllocPtr and GlobalFreePtr macros which have direct mapping to NT functions.

Another issue with porting to NT is the use of LPSTR casts when doing address arithmetic such as is used to calculate the offset of DIB bits in a packed DIB. It is better to use LPBYTE since this points to an 8 bit object in both 16 and 32 bit Windows environments. LPSTR points to a 16 bit object in Windows NT.

End.