Game Programming Genesis
Part III : Tracking Your Window and Using GDI

by Joseph "Ironblayde" Farrell

Introduction

If you've been with me for the last two articles, you've probably been asking yourself when I'm going to show you something useful. Well, the wait is over! Today I'll be showing you the basics of Windows GDI (Graphical Device Interface), and a few other things along the way, like responding to user input and dealing with some more of the messages that Windows generates. As far as actually displaying graphics, I'm going to go over three basic topics: showing text, plotting pixels, and displaying bitmaps. Before getting into too much of that though, I'm going to cover several more Windows messages in detail so you will be sure to know what's going on when the user starts messing with things. They always do. :)

As always, you need only a basic knowledge of the C language, and the information that was covered in previous articles of this series. Since this article will enable you to make some working graphical demos, there is a sample program available along with the article. The code used for this program was written and compiled in Visual C++, but it is simple enough that you shouldn't have to change it to get it working with other compilers. All right, enough with the disclaimers, and on to the fun stuff!

Device Contexts

In the first article in this series, we defined and registered a window class. One of the lines in that definition, giving the window's capabilities, was this:

```
sampleClass.style = CS_DBLCLKS | CS_OWNDC |
                    CS_HREDRAW | CS_VREDRAW;   // standard settings
```

Three of those attributes are fairly self-explanatory, but the other -- `CS_OWNDC` -- requires some explanation. If you recall, I told you that this attribute allowed for the window to have its own unique device context, and that device contexts would not be covered just yet. Well, grasshopper, the time has come.

A device context is a structure that represents a group of graphic objects and their attributes, as well as some output device and its attributes and settings. Using device contexts allows you to manipulate graphics in a very straightforward manner, without having to worry about a lot of low-level details. Windows GDI is a graphics-rendering system which takes Windows graphics calls and passes the information to the appropriate device driver. To make use of GDI graphics, you must use device contexts. Thankfully, it's very easy to do. You can get a device context for a window using a simple function call:

```
HDC GetDC( HWND hWnd   // handle to a window );
```

That looks pretty harmless, doesn't it? All you do is pass a handle to the window for which you want a device context (or DC), and the return value is a handle to that device context. If you pass `NULL`, the handle returned is for a DC to the entire screen. If the function call fails, the return value is `NULL`.

Now is a good place to mention that device contexts are a little more general than dealing with graphics calls only. The type of DC we'll be talking about is called a display device context, because it deals with displaying graphics. In addition, there are printer device contexts, which use a printer as the output device; memory device contexts, which allow for manipulation of bitmap data; and information device contexts, for retrieving data for a specified device. Don't worry if this all sounds complicated. It's Windows -- its primary function is to confuse people. :) Once we get into some code, I think you'll find that it's actually not that difficult.

When you're finished with a device context, you have to release it. This frees up any memory that was being used by the object -- you'll come across the concept of releasing objects a lot more in the future. Once again, this is done by using a simple function call:

```
int ReleaseDC(
     HWND hWnd,  // handle to window
     HDC hDC     // handle to device context );
```

The return value is 1 if the DC was successfully released, or 0 if something went wrong. The parameters are self-explanatory, but I'll list them here anyway.

`HWND hWnd`: This is the handle to the window which is referred to by the DC you're trying to release. If you have a DC for the whole desktop, pass `NULL`.

`HDC hDC`: The handle to the device context you want to release.

Before we get into doing some graphics displays with device contexts and GDI, I want to talk about some of the important messages you'll encounter when creating a windowed application. The four messages I want to cover briefly are `WM_MOVE`, `WM_SIZE`, `WM_ACTIVATE`, and `WM_PAINT`.

Tracking the Status of Your Window

The first two are relatively simple. `WM_MOVE` is called whenever the window is moved by the user. The new window coordinates are stored in `lparam`. (Remember, messages are further specified by the contents of `lparam` and `wparam`, which are parameters received by your message-handling function.) The low word of `lparam` is the x-coordinate of the upper-left corner of the window's client area. The high word of `lparam` is the y-coordinate.

The WM_SIZE message is sent when the window is resized. Like the WM_MOVE message, its parameterization is held in lparam. The low word is the client area's width, and the high word is its height. But unlike WM_MOVE, the wparam parameter also holds some significant. It can take any of the following values:

SIZE_MAXHIDE	Some other window has been maximized.
SIZE_MAXIMIZED	Window has been maximized.
SIZE_MAXSHOW	Some other window has been restored.
SIZE_MINIMIZED	Window has been minimized.
SIZE_RESTORED	Window has been resized, but neither maximized nor minimized.

When I'm writing windowed applications, I usually like to keep a few global variables that give the window's current position and size. If these variables were called xPos, yPos, xSize, and ySize, you'd handle the WM_SIZE and WM_MOVE messages something like this:

```
if (msg == WM_SIZE)
{
  xSize = LOWORD(lparam);
  ySize = HIWORD(lparam);
}

if (msg == WM_MOVE)
{
  xPos = LOWORD(lparam);
  yPos = HIWORD(lparam);
}
```

Next up is the WM_ACTIVATE message, which tells you when a new window becomes the active window. This can be useful because you may not want to be processing all of your program's logic if some other application has the focus. Sometimes, such as in writing fullscreen DirectX programs, ignoring the WM_ACTIVATE message can cause your program to experience a fatal error by doing something it's not supposed to be doing. In any case, it's good to watch the WM_ACTIVATE messages and take action accordingly.

The WM_ACTIVATE message is sent to both the window being activated, and the window being deactivated. You can determine which is the case by looking at the low word of wparam. It will be set to one of three possible values:

WA_CLICKACTIVE	Window was activated by a mouse click.
WA_ACTIVE	Window was activated by some other means (keyboard, function call, etc.)
WA_INACTIVE	Window was deactivated.

For dealing with this message, I'll keep another global variable called bFocus, and change its value when a WM_ACTIVATE message is received. The code would look something like this:

```
if (msg == WM_ACTIVATE)
{
  if (LOWORD(wparam) == WA_INACTIVE)
    focus = FALSE;
  else
    focus = TRUE;

  // tell Windows we handled it
  return(0);
}
```

There are two related messages called WM_KILLFOCUS and WM_SETFOCUS, which a window receives immediately before it loses or gains the keyboard focus, respectively. Since it's possible for no window to have the keyboard focus, I suggest using the WM_ACTIVATE message to track your window's status. Now, on to the biggie.

The WM_PAINT Message

A window receives this important message when part of its client area has become invalidated. Suppose your program doesn't have the focus, and the active window is on top of your window. If the user moves that active window, it's going to reveal a part of your window. Since that part of the window needs to be refreshed, it is said to be invalidated. To handle this, there are a couple of things you can do. The first involves a pair of functions designed exclusively for use with the WM_PAINT message. The first is BeginPaint(). Here's the prototype:

```
HDC BeginPaint(
     HWND hwnd,             // handle to window
     LPPAINTSTRUCT lpPaint  // pointer to structure for paint information );
```

Before I tell you exactly what the return value is, let's look at the parameters:

HWND hwnd: This is a handle to the window which needs repainting. You should be used to seeing this parameter by now, right?

LPPAINTSTRUCT lpPaint: Here's the important one. This is a pointer to a PAINTSTRUCT structure, which contains all sorts of information about the area to be painted.

And before we go on, I should show you exactly what a PAINTSTRUCT looks like...

```
typedef struct tagPAINTSTRUCT { // ps
```

```
        HDC  hdc;
        BOOL fErase;
        RECT rcPaint;
        BOOL fRestore;
        BOOL fIncUpdate;
        BYTE rgbReserved[32];
    } PAINTSTRUCT;
```

And the members of the structure are as follows:

HDC hdc: Aha! I knew there was some reason we went over device contexts, even if it took awile to get here. This is a DC that represents the invalidated area -- the area that needs to be painted.

BOOL fErase: This specifies whether or not the application should erase the background. If set to FALSE, the system has already deleted the background. Remember in our window class when we defined a black brush as the background? This will cause the system to automatically erase the invalidated area with that black brush.

RECT rcPaint: This is the most important member, as it tells you the rectangle that needs to be repainted in order to cover the whole invalidated area. I'll show you the RECT structure in just a bit.

BOOL fRestore, BOOL fIncUpdate, BYTE rgbReserved[32]: Good news! These are reserved and are used by Windows, so you and I don't have to worry about them. :)

Now that I've showed this to you, I can tell you just what BeginPaint() is accomplishing. It actually does three things. First, it validates the window again, so that another WM_PAINT message will not be generated unless the window becomes invalidated again. Second, if your window class has a background brush defined, like ours does, it paints the affected area with that brush. Third, it returns a handle to a device context which represents the area needing to be painted. That area, as we saw, is defined by the important RECT structure:

```
    typedef struct _RECT {
        LONG left;
        LONG top;
        LONG right;
        LONG bottom;
    } RECT;
```

You've already figured out that this structure represents a rectangle, but there is one thing that needs to be said about it. RECTs are upper-left inclusive, but lower-right exclusive. What does that mean? Well, let's say you define a RECT like this:

```
    RECT myRect = {0, 0, 5, 5};
```

This RECT includes the pixel at (0, 0), but it stops short of (5, 5), so that the lower-right corner of the area described by this rectangle is actually at (4, 4). It doesn't seem to make much sense at first, but you'll get used to the idea.

Now, remember what I said about using device contexts? Once you're done using one, you have to release it. In this case, you use the EndPaint() function. Each call to BeginPaint(), which should only be made in response to a WM_PAINT message, must have a matching EndPaint() function to release the DC. Here's the function:

```
    BOOL EndPaint(
         HWND hWnd,  // handle to window
         CONST PAINTSTRUCT *lpPaint  // pointer to structure for paint data );
```

The function returns TRUE or FALSE indicating its success or failure, respectively, and takes two simple parameters:

HWND hWnd: Just the handle to the window. Again.

CONST PAINTSTRUCT *lpPaint: A pointer to the PAINTSTRUCT containing the information about the area in question. Don't let the CONST confuse you. It's just there to denote and ensure that the function does not alter the contents of the structure.

For the record, the other way you can validate a window is with a call to ValidateRect(). If you want to do everything manually instead of letting BeginPaint() handle it, that's fine. There may be some cases where this is necessary. So here's the prototype:

```
    BOOL ValidateRect(
         HWND hWnd,  // handle of window
         CONST RECT *lpRect  // address of validation rectangle coordinates );
```

The return value is TRUE or FALSE for success or failure, and the parameters are easy to figure out:

HWND hWnd: Are you getting tired of seeing this yet? :)

CONST RECT *lpRect: This is a pointer to the RECT to validate. Again, you don't need to declare it as a constant; the CONST is just to make sure the function doesn't go changing things on you. If you pass NULL, the entire client area is validated.

Now, to wrap up our discussion of this message, I'll show you the framework for handling a WM_PAINT message. This would be somewhere in your message handler, as usual. I'm assuming here that we have a global variable called hMainWindow that is the handle to our window.

```
if (msg == WM_PAINT) {
   PAINTSTRUCT ps;  // declare a PAINTSTRUCT for use with this message
   HDC hdc;         // display device context for graphics calls
   hdc = BeginPaint(hMainWindow, &ps);  // validate the window

   // your painting goes here!

   EndPaint(hMainWindow, &ps);  // release the DC

  // tell Windows we took care of it
  return(0);
}
```

The only part of that code that probably doesn't make sense is the place where I have commented, "Your painting goes here!" Well, if you want your window to be refreshed with something other than your window class's default brush, you have to do it yourself, and that involves some graphics work that you haven't seen yet. Never fear, we'll get there in just a minute! While we're on the topic of messages, though, there's something I need to explain.

Closing Your Application

There are three messages that seem to be pratically identical, and all deal with closing things out. They are `WM_DESTROY`, `WM_CLOSE`, and `WM_QUIT`. They're similar, but you need to know the difference! `WM_CLOSE` is sent when a window or application should be closing. When you receive a `WM_CLOSE message`, it's a good place to ask the user if they're sure they want to quit, if you want to do it. You know those little message boxes that are always popping up on your screen when errors or notifications occur? Well, they're easy to create. In addition to serving many functions in the final program, they're also handy for reporting debug information. The call to create your very own message box is pretty simple:

```
int MessageBox(
     HWND hWnd,          // handle of owner window
     LPCTSTR lpText,     // address of text in message box
     LPCTSTR lpCaption,  // address of title of message box
     UINT uType          // style of message box );
```

The parameters, especially the last one, require some explanation:

`HWND hWnd`: Sooner or later we'll get to a function that doesn't have this, I promise!

`LPCTSTR lpText`: This is the text that will appear in the message box. As always, you can use escape sequences like `\n` to format the output a little if you want.

`LPCTSTR lpCaption`: This is the text appearing in the message box's caption bar.

UINT uType: You can combine several different flags in order to create this parameter, which defines what kind of message box it will be. There are a lot of MB_ constants you can use, and you can combine any number of them with the | operator. Here's a list of the useful ones:

Button Definition Flags	
MB_ABORTRETRYIGNORE	Creates a box with "Abort," "Retry," and "Ignore" buttons.
MB_OK	Creates a box with an "OK" button.
MB_OKCANCEL	Creates a box with "OK" and "Cancel" buttons.
MB_RETRYCANCEL	Creates a box with "Retry" and "Cancel" buttons.
MB_YESNO	Creates a box with "Yes" and "No" buttons.
MB_YESNOCANCEL	Creates a box with "Yes," "No," and "Cancel" buttons.
Icon Definition Flags	
MB_ICONEXCLAMATION	Adds an exclamation point icon to the box.
MB_ICONINFORMATION	Adds an information icon to the box.
MB_ICONQUESTIION	Adds a question mark icon to the box.
MB_ICONSTOP	Adds a stop sign icon to the box.
Default Button Flags	
MB_DEFBUTTON1	Defines the first button as the default.
MB_DEFBUTTON2	Defines the second button as the default.
MB_DEFBUTTON3	Defines the third button as the default.
MB_DEFBUTTON4	Defines the fourth button as the default.
Other Flags	
MB_HELP	Adds a help button to the box. A WM_HELP message is generated if the user chooses it or presses F1.
MB_RIGHT	Message box text is right-justified.
MB_TOPMOST	Sets the message box to always be the topmost window.

I don't know about you, but I'm starting to think that Microsoft has a programmer who does nothing but write #define statements all day! Now, the return value is 0 if the box could not be created. Otherwise, the result is one of the following:

IDABORT	"Abort" button was selected.
IDCANCEL	"Cancel" button was selected.
IDIGNORE	"Ignore" button was selected.
IDNO	"No" button was selected.
IDOK	"OK" button was selected.

IDRETRY	"Retry" button was selected.
IDYES	"Yes" button was selected.

Those lists were so long I almost forgot what we were originally talking about. Anyway, when you receive a WM_CLOSE message, you can do two things. First, you can allow the default handler to return a value. If you do this, the application or window will close as planned. However, if you return 0, the message will have no effect. This is the basis of the following bit of code:

```
if (msg == WM_CLOSE) {
  if (MessageBox(hMainWindow,
                 "Are you sure want to quit?",
                 "Notice",
                 MB_YESNO | MB_ICONEXCLAMATION) == IDNO)
    return(0);

  // otherwise, let the default handler take care of it
}
```

Now, WM_DESTROY is a bit different. It is sent when a window is being closed. By the time you get a WM_DESTROY message, the window it applies to has already been deleted from view. If the main window closes, that does not necessarilly end the application. It will keep running, but without a window. However, when a user closes the main window, they almost always mean to close the application, so you have to post a WM_QUIT message when you receive WM_DESTROY if you want the application to end. You could use PostMessage(), but since this is a special case, there's a special function for it:

```
VOID PostQuitMessage(int nExitCode);
```

The parameter is an exit code that your application returns to Windows. Remember, WinMain() returns an int, not a void. The nExitCode parameter also becomes the wparam member of the WM_QUIT message that results. WM_QUIT represents a request to close the application, so when you get one, you should end your main loop and return wparam to Windows. Here's an example of what a simplified WinMain() function might look like with this in place:

```
int WinMain(HINSTANCE hinstance,
            HINSTANCE hPrevInstance,
            LPSTR     lpCmdLine,
            int       nCmdShow)
{
  // initialization stuff goes here

  // main loop - infinite!
  while (TRUE)
```

```
      {
        // check the message queue
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
          if (msg.message == WM_QUIT)  // exit main loop on WM_QUIT
            break;

          TranslateMessage(&msg);
          DispatchMessage(&msg);
        }

        // main program logic goes here
      }

      // perform any shutdown functions here - releasing objects and such

      return(msg.wparam);  // return exit code to Windows
    }
```

Sorry about all that stuff, but it's necessary to make sure your program behaves correctly instead of causing errors for Windows -- like that ever happens! Now instead of making you any more impatient with me than you probably already are, let's look at some basic GDI graphics.

Plotting Pixels

At last! Plotting pixels with GDI is a cinch as long as you've got a display device context to work with. Remember, calling `GetDC()` does this for you. To plot a pixel, not surprisingly, you call `SetPixel()`:

```
COLORREF SetPixel(
     HDC hdc,           // handle to device context
     int X,             // x-coordinate of pixel
     int Y,             // y-coordinate of pixel
     COLORREF crColor   // pixel color );
```

The return type is something we haven't encountered yet, a `COLORREF`. This is not a structure, but a 32-bit value in the form `0x00bbggrr`, where `bb` is an 8-bit value for the blue component, `gg` is green, and `rr` is red. The high byte is unused and is always set to zero. Let's take a look at the parameters for `SetPixel()`:

`HDC hdc`: This is a device context for your window that you should obtain with a call to `GetDC()`. You only need to call `GetDC()` once, and then you can use it for any number of these functions. Don't get a new DC every time you want to plot a pixel!

`int X, Y`: The x- and y-coordinates of the pixel. These are in client coordinates, meaning that (0, 0) represents the upper-left corner of your window's client area, not the upper-left corner of the screen.

COLORREF crColor: This is the color you want to set the pixel to. To do this, it's easiest to use the RGB() macro, which takes values for red, green, and blue -- in that order -- between 0 and 255. SetPixel() will choose the closest available color to the one you have specified.

If the function succeeds, the return value is the color that the pixel was set to. This may not always be exactly the COLORREF you pass if you're working in less than 24-bit color; Windows will choose the closest match. If the function fails, it returns -1. As an example, if you want to set the upper-left corner of your client area to white, you'd use the following call:

```
SetPixel(hdc, 0, 0, RGB(255, 255, 255));
```

This call assumes you've gotten a display device context named hdc. Pretty easy, hey? There's one other way to do it that's just a tad faster. Here's the function:

```
BOOL SetPixelV(
     HDC hdc,           // handle to device context
     int X,             // x-coordinate of pixel
     int Y,             // y-coordinate of pixel
     COLORREF crColor   // new pixel color );
```

The parameters are all the same. The return value is simply TRUE or FALSE for success or failure. SetPixelV() is slightly faster since it doesn't need to return the actual color that was used to plot. You'll probably never even notice the difference, unless you're using it thousands of times per frame, but if you don't need the extra information SetPixel() provides, there's no reason not to take the slightly increased performance, right?

The only other thing you need to know about plotting pixels is how to read the value of a pixel that's already been plotted. It's no problem; a quick call to GetPixel() does the job for you:

```
COLORREF GetPixel(
     HDC hdc,   // handle to device context
     int XPos,  // x-coordinate of pixel
     int nYPos  // y-coordinate of pixel );
```

The return value is obviously the color of the pixel at the given coordinates. If the coordinates specified are outside the clipping region (the area represented by the device context), the return value is CLR_INVALID. The parameters are the same as for SetPixel(): a device context to use, and the coordinates to operate on. That's it for plotting pixels. Now let's have a look at GDI's text-rendering functions.

GDI Text Functions

There are two functions for actually plotting text that you need to be concerned with. The simpler of the two is TextOut(), as shown here:

```
BOOL TextOut(
     HDC hdc,           // handle to device context
     int nXStart,       // x-coordinate of starting position
     int nYStart,       // y-coordinate of starting position
     LPCTSTR lpString,  // pointer to string
     int cbString       // number of characters in string );
```

By now we've seen enough BOOL-returning functions to know what that means: TRUE for success, FALSE for failure. The parameters are:

HDC hdc: The device context to use.

int nXStart, nYStart: These are the coordinates of the starting point for the text, called the reference point. By default, this is the upper-left corner of the rectangular area occupied by the string. You can change this setting, as we'll see in just a bit.

LPCTSTR lpString: The text to print out. Since the number of characters is given in the final parameter, this string does not need to be null-terminated.

int cbString: This is the length of the string, in characters.

TextOut() uses the current settings for text color, background color, and background type. Before looking at the other, more complicated text-rendering function, let's take a look at the functions you can use to control the colors being used.

```
COLORREF SetTextColor(
     HDC hdc,           // handle to device context
     COLORREF crColor   // text color );

COLORREF SetBkColor(
     HDC hdc,           // handle of device context
     COLORREF crColor   // background color value );
```

SetTextColor() sets the active text color, and SetBkColor() sets the active background color. The parameters are obviously the device context to apply the settings to, and the colors to use. Since these are COLORREFs, remember that you can use the RGB() macro for specifying your colors. Each function returns the previous value of the attribute it deals with. For instance, if you call SetTextColor(hdc, RGB(255, 0, 0)), the return value will be the active color that was being used before you turned it red. Finally, to set the background type, use SetBkType() as shown:

```
int SetBkMode(
     HDC hdc,      // handle of device context
     int iBkMode   // flag specifying background mode );
```

The device context parameter we've seen before, but the other, iBkMode, can take one of two values: TRANSPARENT or OPAQUE. If set to TRANSPARENT, any text you plot will not disturb the background around the text itself. If set to OPAQUE, plotting text will cause the rectangular region surrounding that text to be filled with the active background color. The return value of SetBkMode() is simply the previous background mode.

One more thing about TextOut(). I said you could change the way the reference point is interpreted, and the way to do it is by using SetTextAlign(), whose prototype is shown below.

```
UINT SetTextAlign(
     HDC hdc,     // handle to device context
     UINT fMode   // text-alignment flag );
```

The parameters are:

HDC hdc: The device context again. No surprises here.

UINT fMode: A flag or set of flags (logically combined with |) that determine the meaning of the reference point in a call to TextOut(). Only one flag can be selected from those affecting horizontal and vertical alignment, and only one of the two flags affecting use of the current position can be used. The flags are:

TA_BASELINE	The reference point will be on the baseline of the text.
TA_BOTTOM	The reference point will be on the bottom edge of the bounding rectangle.
TA_TOP	The reference point will be on the top edge of the bounding rectangle.
TA_CENTER	The reference point will be aligned horizontally with the center of the bounding rectangle.
TA_LEFT	The reference point will be on the left edge of the bounding rectangle.
TA_RIGHT	The reference point will be on the right edge of the bounding rectangle.
TA_NOUPDATECP	The current position is not updated by a call to a text output function. The reference point is passed with each call.
TA_UPDATECP	The current position is updated by each call to a text output function, and is used as the reference point.

The default setting is TA_LEFT | TA_TOP | TA_NOUPDATECP. If you set TA_UPDATECP, subsequent calls to TextOut() will ignore the nXStart and nYStart parameters, and render the text where the last call left off. Now that that's out of the way, let's look at the bells-and-whistles version of

TextOut(), called DrawText():

```
     int DrawText(
          HDC hDC,          // handle to device context
          LPCTSTR lpString, // pointer to string to draw
          int nCount,       // string length, in characters
          LPRECT lpRect,    // pointer to struct with formatting dimensions
          UINT uFormat      // text-drawing flags );
```

This one gets a bit complicated. Since DrawText() formats text, possibly to multiple lines, the return value is the height of the text in pixels, or 0 if the function fails. Let's take a look at the parameters, shall we?

HDC hDC: Nothing new here; it's just our good buddy the DC.

LPCTSTR lpString: This is the string to print.

int nCount: This is the length of the string in characters.

LPRECT lpRect: Here's where things start to get a bit different. DrawText() does several different methods of formatting, including word wrapping, so you must specify a RECT within which to format the text, rather than simply passing coordinates.

UINT uFormat: For this, you can use one or more (logically combined with |) of a long list of flags that represent different methods of formatting. I'll show you a few of them.

DT_BOTTOM	Justifies text to the bottom of the RECT. This must be combined with DT_SINGLELINE.
DT_CALCRECT	Calculates the RECT needed to hold the text. If the text is on multiple lines, DrawText() uses your RECT's width and alters the height. If the text is on a single line, DrawText() alters your RECT's width. In both cases, DrawText() adjusts the RECT but does not actually draw the text.
DT_CENTER	Centers text within the RECT you specify.
DT_EXPANDTABS	If the string contains any tabs (\t), this attribute causes DrawText() to expand them. The default is eight spaces per tab.
DT_LEFT	Left-justifies the text.
DT_NOCLIP	Draws without clipping. This speeds up DrawText() a bit.
DT_RIGHT	Right-justifies the text.
DT_SINGLELINE	Displays text on a single line only. Carriage returns and line feeds do not overrule this attribute.

DT_TABSTOP	Alters the number of spaces per tab. The number of spaces per tab must be specified in bits 15-8 (the high byte of the low word) of uFormat. Again, the default setting is eight.
DT_TOP	Justifies text to the top of the RECT. This must be combined with DT_SINGLELINE.
DT_VCENTER	Centers the text vertically within the RECT. This must be combined with DT_SINGLELINE.

There are more of these flags, but you get the idea. All in all, this constitutes a pretty powerful text rendering system, but remember, all those cool features are going to slow the function down. You can usually get by just fine by using TextOut(). That takes care of the text rendering system, so let's do something a little more exciting.

Displaying Bitmaps With GDI

Remember when I told you that bitmaps are easy to work with, because they're native to Windows? Well now we're going to find out just how easy it is. :) There are four basic steps to displaying a bitmap with GDI:

1. Get a device context to your window.
2. Obtain a handle to the bitmap.
3. Create a device context for the bitmap.
4. Copy the image from one device context to the other.

You already know how to do the first step. I alluded to the second one last time, but didn't go over it. I said that there was a function called LoadBitmap() that retrieves a handle to a bitmap resource. However, this function is obsolete now; it has been superseded by LoadImage(), which is much more flexible. So that's what we'll be using. Here she is:

```
HANDLE LoadImage(
     HINSTANCE hinst,   // handle of the instance containing the image
     LPCTSTR lpszName,  // name or identifier of image
     UINT uType,        // type of image
     int cxDesired,     // desired width
     int cyDesired,     // desired height
     UINT fuLoad        // load flags );
```

The function returns NULL if it fails. Otherwise, you get a handle to the bitmap, which can either be loaded from a resource or from an external file. Notice that since this function can be used for bitmaps, cursors, or icons, the return type is simply HANDLE. In Visual C++ 6.0, you'll need to include a typecast to HBITMAP or the compiler will become angry with you. Here are the parameters for the function:

HINSTANCE hinst: This should be the instance of your application if you're loading a resource, or NULL if you want to load from an external file.

LPCTSTR lpszName: This is either the resource identifier -- remember to use MAKEINTRESOURCE() if you're using numerical constants -- or the full filename of the image you want to load.

UINT uType: Depending on what you want to load, this should be set to either IMAGE_BITMAP, IMAGE_CURSOR, or IMAGE_ICON.

int cxDesired, cyDesired: These are the desired dimensions for the image to be loaded. If you set them to zero, the image's actual dimensions will be used.

UINT fuLoad: Like everything else we've done today, this is one or more of a series of flags which can be logically combined with the | operator. Here are the useful flags:

LR_CREATEDIBSECTION	If uType is IMAGE_BITMAP, this causes the function to return a DIB section rather than a compatible bitmap. (DIB stands for device-independent bitmap.) This basically means to use all of the bitmap's own properties rather than making it conform to the properties of the display device.
LR_DEFAULTSIZE	For icons and cursors, if cxDesired and cyDesired are set to 0, this flag causes the system metric values for icons and cursors to be used, rather than the actual dimensions of the image.
LR_LOADFROMFILE	You must specify this flag if you want to load from a file rather than a resource.

For loading bitmaps you should use LR_CREATEDIBSECTION, and LR_LOADFROMFILE if it is appropriate. Now that you have obtained a handle to your image, you must create a device context and load the bitmap into it. The first step is taken by calling CreateCompatibleDC(), as follows:

```
HDC CreateCompatibleDC(HDC hdc);
```

The parameter is a DC with which to make the new DC compatible. If you pass NULL, the DC will be made compatiable with the display screen, which is what we want. The return value is a handle to a memory device context -- **not** a display device context! This means that the contents of this DC won't be visible. If the function fails, the return value is NULL. Now, to get the bitmap into the memory device context, we use this:

```
HGDIOBJ SelectObject(
     HDC hdc,          // handle to device context
     HGDIOBJ hgdiobj   // handle to object );
```

The type HGDIOBJ is more general than our HBITMAP, so never fear, they're compatible without any tricks on our part. Here are the parameters:

HDC hdc: This is a handle to the device context which we want to fill with an object. For loading bitmaps, this must be a memory device context.

HGDIOBJ hgdiobj: And this is a handle to that object. This function is used with bitmaps, brushes, fonts, pens, and regions; but the only one that concerns us is bitmaps.

The return value is a handle to the object that is being replaced in the DC, or NULL if an error occurs. The return values are different for regions, but like I said, we don't care about regions. :)

Now you've got a bitmap loaded into a DC, and you need only take the last step: copying the contents of the memory device context to our display device context. However, it's necessary to obtain some information about the bitmap, such as its dimensions, which must be used in the function call that will display the image. For that, we need the GetObject() function, which is used for obtaining information about graphical objects such as bitmaps.

```
int GetObject(
     HGDIOBJ hgdiobj,  // handle to graphics object of interest
     int cbBuffer,     // size of buffer for object information
     LPVOID lpvObject  // pointer to buffer for object information );
```

The return value is the number of bytes successfully obtained, or 0 for function failure. The parameters for the function are the following:

HGDIOBJ hgdiobj: The handle to the graphics object we want information on. In this case, pass the handle to the bitmap we loaded.

int cbBuffer: This is the size of the structure receiving the information. In the case of loading bitmaps, the receiving structure is of type BITMAP, so set this to sizeof(BITMAP).

LPVOID lpvObject: Pass the address of the structure receiving the information.

You need to define a variable of type BITMAP, and with a quick call to the GetObject() function, you'll have the information you need. Since the BITMAP structure is new to us, I'll show you what it looks like:

```
typedef struct tagBITMAP {  // bm
     LONG   bmType;
     LONG   bmWidth;
     LONG   bmHeight;
     LONG   bmWidthBytes;
     WORD   bmPlanes;
     WORD   bmBitsPixel;
     LPVOID bmBits;
} BITMAP;
```

There aren't too many members to this thing, and we're really only interested in two of them, but I'll list them all here anyway.

`LONG bmType`: This is the bitmap type and must be set to zero. Useful, isn't it?

`LONG bmWidth, bmHeight`: These are the two we're after -- the width and height of the bitmap, in pixels.

`LONG bmWidthBytes`: Specifies the number of bytes in each line of the bitmap. Note that the number of bytes per pixel can be obtained by dividing this value by `bmWidth`.

`LONG bmPlanes`: This is the number of color planes.

`LONG bmBitsPixel`: This is the number of bits required to represent one pixel. It would appear that the note I left about figuring this out is useless. :)

`LPVOID bmBits`: If you want to access the actual image data, this is a pointer to the bit values for the bitmap.

All right, almost done! Now we have the bitmap in a memory device context, and we know its dimensions. All we have to do is copy it from one DC to the other, and that only takes a single function call. See, I told you bitmaps were easy to deal with. There are actually two options you can use here. I'll show you both of them.

```
BOOL BitBlt(
     HDC hdcDest, // handle to destination device context
     int nXDest,  // x-coordinate of destination rectangle's upper-left corner
     int nYDest,  // y-coordinate of destination rectangle's upper-left corner
     int nWidth,  // width of destination rectangle
     int nHeight, // height of destination rectangle
     HDC hdcSrc,  // handle to source device context
     int nXSrc,   // x-coordinate of source rectangle's upper-left corner
     int nYSrc,   // y-coordinate of source rectangle's upper-left corner
     DWORD dwRop  // raster operation code );
```

The return value is `TRUE` or `FALSE` based on whether the function succeeds. You've seen that plenty of times before. There are a lot of parameters, but most of them are pretty easy to figure out.

`HDC hdcDest`: The destination device context handle. In our case, this will be the display device context for our window.

`int nXDest, nYDest`: The coordinates of the upper-left hand corner of the region where the bitmap will end up. Remember that for our DC, these coordinates are client coordinates -- relative to the client area of our window.

`int nWidth, nHeight`: These are the width and height of the destination and source rectangles, since this function doesn't perform scaling. Pass the dimensions of the bitmap.

`HDC hdcSrc`: This is the source device context handle. In our case, this is the memory device context that our bitmap is currently residing in.

`int nXSrc, nYSrc`: The x- and y- coordinates of the source rectangle's upper-left corner. In this case you would use (0, 0), but this may not always be the case, depending on what you're using the source DC for, or if you only want to copy a part of the image.

`DWORD dwRop`: There are a lot of operation codes you can use here, most of them dealing with Boolean operations on the data in the two device contexts. The only one we're interested in is `SRCCOPY`, which copies the contents of the source DC directly to the destination DC.

That's about all there is to it! The other option you have is to use `StretchBlt()`, which requires you to specify the width and height for both the source and destination rectangles. `StretchBlt()` then scales the image in the source DC to fit in the rectangle specified on the destination DC. This can be useful for scaling images, but as always, since `BitBlt()` has fewer capabilities to worry about, it's faster. Here is the prototype for `StretchBlt()`:

```
BOOL StretchBlt(
     HDC hdcDest,      // handle to destination device context
     int nXOriginDest, // x-coordinate of upper-left corner of dest. rectangle
     int nYOriginDest, // y-coordinate of upper-left corner of dest. rectangle
     int nWidthDest,   // width of destination rectangle
     int nHeightDest,  // height of destination rectangle
     HDC hdcSrc,       // handle to source device context
     int nXOriginSrc,  // x-coordinate of upper-left corner of source rectangle
     int nYOriginSrc,  // y-coordinate of upper-left corner of source rectangle
     int nWidthSrc,    // width of source rectangle
     int nHeightSrc,   // height of source rectangle
     DWORD dwRop       // raster operation code );
```

The parameters are basically the same as those in `BitBlt()`, so I shouldn't have to go through them again. The raster operation codes for `StretchBlt()` are the same as those for `BitBlt()`; just set it to `SRCCOPY`. Now, the last thing you need to know is a little bit of cleanup. Creating a DC like we did for this example is not quite like getting a DC for the display device. Instead of calling `ReleaseDC()`, you must call `DeleteDC()`. It looks basically the same:

```
BOOL DeleteDC(HDC hdc);
```

The parameter is simply the device context to delete, and the return value is a `BOOL` again, and we all know what that means, right? All right, are you feeling pretty good about all this stuff? Just for the sake of tying all these steps together, I'll show you a function you can use for loading and displaying a bitmap resource using all GDI functions. For this example, I'm assuming you have saved the handle to the instance of the application in a global called hinstance.

```
int ShowBitmapResource(HDC hDestDC, int xDest, int yDest, int nResID)
{
  HDC hSrcDC;            // source DC - memory device context
  HBITMAP hbitmap;       // handle to the bitmap resource
  BITMAP bmp;            // structure for bitmap info
  int nHeight, nWidth;   // bitmap dimensions

  // first load the bitmap resource
  if ((hbitmap = (HBITMAP)LoadImage(hinstance, MAKEINTRESOURCE(nResID),
                                    IMAGE_BITMAP, 0, 0,
                                    LR_CREATEDIBSECTION)) == NULL)
    return(FALSE);

  // create a DC for the bitmap to use
  if ((hSrcDC = CreateCompatibleDC(NULL)) == NULL)
    return(FALSE);

  // select the bitmap into the DC
  if (SelectObject(hSrcDC, hbitmap) == NULL)
    return(FALSE);

  // get image dimensions
  if (GetObject(hbitmap, sizeof(BITMAP), &bmp) == 0)
    return(FALSE);

  nWidth = bmp.bmWidth;
  nHeight = bmp.bmHeight;

  // copy image from one DC to the other
  if (BitBlt(hDestDC, xDest, yDest, nWidth, nHeight, hSrcDC, 0, 0,
             SRCCOPY) == NULL)
    return(FALSE);

  // kill the memory DC
  DeleteDC(hSrcDC);

  // return success!
  return(TRUE);
}
```

One Last Thing

You may not realize it, but you now have enough knowledge of Windows programming to create a working game based on Windows GDI! You can create the window, and you can show the graphics. The game logic is just the same old C you've come to know and love. You can even use the mouse by processing the messages it generates. There's just one thing you're missing, and even though it's off-topic for this article, I can't leave without mentioning it, and that's keyboard support. Windows has a great function for determining the state of the keys on the keyboard: `GetAsyncKeyState()`. It returns a 16-bit value, the high bit of which indicates whether the key is currently depressed or not. Here's the prototype:

```
SHORT GetAsyncKeyState(int vKey);
```

The parameter is an identifier for a key on the keyboard, and takes constants beginning with `VK_` for "virtual key." Some of the most common ones are `VK_RETURN`, `VK_ESCAPE`, `VK_UP`, `VK_LEFT`, `VK_RIGHT`, and `VK_DOWN`. You can even use `VK_LBUTTON` and `VK_RBUTTON` for the mouse buttons! How convenient. Just mask out the high bit, and if it's 1, the key is being pressed. A useful macro for doing this that I always use is:

```
#define KEYSTATE(vknum) ((GetAsyncKeyState(vknum) & 0x8000) ? TRUE : FALSE)
```

If you're scratching your head over that, you probably haven't seen the conditional operator (`?`) before. This operator -- the only ternary operator in the C language -- evaluates the expression on its left. If the expression is true, the expression evaluates to the value on the left side of the colon. If the expression is false, the expression evaluates to the value on the right side of the colon. Got it? Cool, hey?

A few notes: first, the method I have just shown you only tells whether the key is up or down. It doesn't tell you when the key was pressed. So if you want to test for discrete keypresses rather than continuous ones, you'll have to make up some logic to do this. `GetAsyncKeyState()` also uses the low bit of its return value to tell whether the key has been pressed since the last call to the function, but that may not be good enough. One solution is to create a table of the 256 possible values the function can take, then call each one every frame, and compare the new values to the old ones.

Closing

Now you've got all it takes for a GDI-based game, and I've taken this article a lot longer than I wanted it to go. Congratulations to both of us. :) Since I covered a lot today, I have developed a sample program for you to look at. It recreates the starfield screensaver that comes with Windows, only in an `.EXE` format and in a window instead of fullscreen. The program illustrates the creation of a window, the processing of several key messages, and using a device context for pixel-plotting. The download includes the source and the compiled program, and is available here. Any questions? I'm always happy to help out. You can reach me by E-mail at ironblayde@aeon-software.com, or on ICQ at UIN #53210499.

Now that we've covered all this Windows stuff, next time I'll be taking you on a magical journey into the wonderful world of DirectX. See you then!

Discuss this article in the forums