GDI SEE GDI DO

part Duex

Graphics Animation in Windows

by Bernard Andrys

Well its next month already and I bet all 3 of my readers are wondering when I'll turn my "Do Nothing Program" into a way cool, time wasting, blow'em'all'up arcade game.

But first, a VERY, VERY important correction to the last article:
As was pointed out by " " using the PASCAL keyword for a function declaration does not make a function pass parameters by reference instead of by value. I read about the purpose of the PASCAL keyword in an article but I really should have tried passing parameters using the PASCAL keyword before claiming that it works. I apologize for any confusion that it may have caused.

...and now back to blowing things up.

In this article, we'll at least get something moving. We're going to put the missile base on the screen and make it move when you press the cursor keys. Lets start with what we'll need to put a missile base on the screen and make it move:

1. A missile base
2. A way to put the missile base in the window.
3. A way to link the keyboard to the drawing of the missile base.

The first requirement is pretty easy. We need to define all the properties that a missile base has:

1. Its appearance
2. Its location
3. How many lives it has left.
4. Whether it can shoot or not. (we're only going to let it shoot one missile at a time)
5. Has it been hit by an invader's missile? (so we know if we should draw an explosion instead of a missile base)

We'll work on the first two properties now and save the rest for later.

The first property, position, is its x and y coordinates. We'll use the variables x and y to represent the missile base's x and y coordinates. Before we use the variables x and y, we'll need to declare them. Declaring a variable tells the C compiler what type of variable is going to be used. We should pick a data type for our variables x and y that is big enough to handle the largest
values that might be assigned to x and y.

So how big are the data types that we have to work with?

Data Type	Size
char	-128 to 127
int	-32,768 to 32,767
unsigned char	0 to 255

unsigned int	0 to 65,535
unsigned int	0 to 4,294,967,295
float	3.4E _ 38
double	1.7E _ 308
void	nothing

There are alot of other data types supported by C compilers such as long int and short int but considering that Win32 and NT are just around the corner you probably don't need the confusion of learning X86 segmentation. (Especially since this IS a beginner's column.) Windows has its own data types as well. For example HBITMAP and HINSTANCE are two data types that are specific to Windows. You would declare a variable as HBITMAP if the variable was going to contain a handle to a bitmap. A handle is a number that a program uses to refer to a file, bitmap, or some other object. You don't normally have to care about the size of Window's own data types like HBITMAP or HINSTANCE because Window's own data types are normally used as return values or parameters for Window's own functions. For example, if we used a function called LoadBitmap() it would return a variable of type HBITMAP. The HBITMAP variable would then be used where ever we want to refer to the actual bitmap.

EXAMPLE:

```
MyFunction()
{
HBITMAP mybitmap;  // declare a variable called mybitmap which is data type HBITMAP

mybitmap = LoadBitmap(hInstance, "bitmap");  // Window's function LoadBitmap()
                                             // returns a variable of type HBITMAP
}
```

Actually, Windows doesn't have its own data types. Data types like HBITMAP are just defined in the file windows.h as an unsigned integer. You declare the variable as HBITMAP instead of unsigned integer because the size of HBITMAP might (and will) change under NT or maybe even in future versions of windows like Windows 4.0. The size of HBITMAP could even change for each CPU that the program is compiled for (this is an NT consideration again).

Now that we know what data types are available, we can pick a data type for x and y. Since our screen is only 300 pixels high and 300 pixels wide we should define x and y as type int. We'll probably have to use the position of the missile base in many places throughout our code so its best the we define x and y as global variables by defining them in the header file "invid.h".

The second property, its appearance, is defined by a bitmap that we'll have represent the missile base. You can create the bitmap of the missile base using any program that can create .bmp files. We're going to use 16 color bitmaps so be sure to save the file in that format. Notice that the bitmap base.bmp has black pixels to its right and left. Since the missile base only moves right and left, and the background is solid black, we can cheat in creating the bitmap animation. Instead of having to make windows properly redraw the background (using WM_PAINT messages), or using a bitmap mask to draw the bitmap without disturbing the background, we can just draw the bitmap to the screen directly. As long as we move the base to the right or left 3 pixels or less at a time (There's a 3 pixel black border on each side of the base.), drawing the base covers over the previous base bitmap that was on the screen.

To use a bitmap in a windows program you first need to define it in your program's .rc file. Here is the contents of the invid.rc file:

```
#include "windows.h"

invid           ICON        invid.ico
base            BITMAP              base.bmp
```

We've added the missle base's bitmap (base.bmp) to the end of the .rc file after the icon file. The first entry, base, will be what we call the bitmap file in the program's source code. The second entry, BITMAP, tells the resource compiler that this is a bitmap resource. The last entry, base.bmp, is the name of the bitmap file that we want to load. So now our program will contain the bitmap as a resource that can be loaded and used by our program. To use the resource, you call the function LoadBitmap(Program_Instance, "bitmap name"). The first parameter to LoadBitmap() is the handle of the program instance. So we'll need to save the hInstance variable that is passed to WinMain() by assigning it to our own variable hInst. That way we can use hInst when we call the LoadBitmap() function. The second parameter is the name of the bitmap that we defined in the .rc file. The LoadBitmap function returns a variable of type HBITMAP that we use whenever we want to refer to the Bitmap. I added a variable called hbBase as type HBITMAP in our header file invid.h.

Fast and Friendly Animation

Now that we've got our missle base, how do we display it onto the screen?

There are two ways we can go about it:

1) Display the missle base at its position every time we get a Timer message.
2) Display the missle base at its position whenever it moves.

The first method would be ideal for multitasking friendliness. However, this method is too slow. At best, windows can only provide 18 timer messages per second. I originally thought that this would be fast enough. After all, tv is at 30 frames/sec, movies run at 24 frames/sec and Microsoft Video files look fine at 15 frames/sec. Unfortunatly it didn't work out that way. 18 frames per second is just too slow for arcade game speed. We want the animation to be as smooth as possible. That means that if the user tapped the right cursor key quickly, the missile base would move 1 pixel to the right. If the user held the cursor key down, the base would move 18 pixels/second across the screen. However, our window is currently 400 pixels wide. So it would take over 22 seconds to move the base from one end of the screen to the other. Twenty-two seconds is a LONG time in arcade style action games. This method just won't work with Windows only sending 18 timer messages per second. We could make this method work if we could get Windows to send us more than 18 timer messages per second. There are ways of doing this using the multimedia timer services but they seem fairly complicated to me. (The real reason is that I've been too lazy to upgrade from Quick C which only supports the Win 3.0 API.)

The second method is much easier to deal with but creates its own problems. Instead of having our program only do something in response to a message, we can instead write our program to run continuously. Unfortunately to use this method in Windows will require some care because we want our program to be friendly to other Windows programs and not stop every other Windows program just because ours is running. To do this we can use Windows' Peekmessage() function to see if Windows as anything else to do. If it does, we'll let Windows take care of the other programs before coming back to our program. This way our program will run continuously, as long as no other program has things to do. An obstacle to using Peekmessage is that you shouldn't keep your program in a Peekmessage loop because it keeps Windows from posting idle messages. Another problem with this method is that since our program runs as fast as the CPU will allow, we will have to do our own timing to keep the animation rate constant no matter what CPU we're running on.

So lets get started:

```
/* invid.c */

#include <windows.h>
#include "invid.h"

//---------------------------------- Global Variables----------------------
//
// In the real source code I keep my global variables in
// "invid.h" but its easier for you to read the source
// code if I put them here.
//

// Declare a structure called decodeWord that will be
// used by WinProc to run a function in response to
// a message
struct decodeWord {
        WORD Code;
        LONG (*Fxn)(HWND, WORD, WORD, LONG); };

// An array called messages made up of decodeWord structures is created.
// The array lists the Windows Message and the function that will be called
// when that message is received.  I've added two messages, WM_KEYDOWN
// and WM_KEYUP.  When WM_KEYDOWN or WM_KEYUP is received,
// the function DoKeydown or DoKeyUp is run.
struct decodeWord messages[] = {
  WM_KEYDOWN, DoKeydown,
        WM_KEYUP, DoKeyup,
        WM_CREATE, DoCreate,
        WM_DESTROY, DoDestroy,
} ;

// Declare a character string that contains the name of the program.
char szAppName [] = "Invid";

// Declare a variable called hInst that contains the data for
// a particular instance of the program.
HANDLE hInst;

// Declare a variable called hWnd that contains a handle to
// the programs window.
HWND hWnd;

// Declare an integer called AnimeStep.  AnimeStep is how
// many pixels a bitmap should be moved during animation.
int AnimeStep = 2;

// Declare two variables of type BOOL (boolean).  That means
// that the variables can be a 1 or a 0.  The variables fRightKeyState
// and fLeftKeyState will be set by the functions DoKeyUp and DoKeyDown.
// The variables will then be used by other functions so that they can know
// the state of the right and left cursor keys.  (I could have used the
```

```
// windows function GetAsyncKeyState() intead of waiting for a keypress
// message to be sent by Windows.  But I wanted the program to be
// written in the standard Windows programming style of waiting for a message.)
BOOL  fRightKeyDown = FALSE, fLeftKeyDown = FALSE;

//  Decalaring a structure called BaseStruct that will contain everything about
// the missile base.  I could have just as easily declared x, y, and Bmp as separate
// variables.  Putting them into a structure groups related variables under one
// title and makes it easy to remember what variable does what.

struct BaseStruct {
// Declare an integer called x that will contain the missile base's x coordinate
        int x ;
// Declare an integer called y that will contain the missile base's y coordinate
        int y ;
// Declare a variable called Bmp that contains a handle
// to the bitmap of the missile base.
        HBITMAP Bmp ;
        };

// Declare a Structure called Base of type BaseStruct (defined above).
// You can now access the missile base's variables x, y and Bmp by using
// Base.x, Base.y, and Base.Bmp.
struct BaseStruct Base;

// ---------------------------------  WinMain ---------------------------------
int PASCAL WinMain (HANDLE hInstance,
        HANDLE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow)
{
        MSG msg ;
        hInst = hInstance;

        if (!hPrevInstance) InitApplication(hInstance);
        InitInstance(hInstance, nCmdShow);


// This is the old message loop from the last program.
/*
        while (GetMessage (&msg, NULL, 0, 0))
        {
                TranslateMessage (&msg) ;
                DispatchMessage (&msg) ;
        }
*/

// This is the new message loop (from Petzold).  Instead of calling GetMessage() to
// retrieve a message from Windows, this message loop calls PeekMessage
// to see if there are any messages waiting.  If the waiting message is
// a WM_QUIT message then break out of the loop and end our program.
// If the waiting message is not a WM_QUIT message then translate and
// dipatch the message.  If there are no waiting messages then run our
// function DoInv().  This message loop will allow other Windows programs to
// run at the same time but it will also keep idle messages from
```

```
// being generated.  We'll have to fix it in a future article to make it more
// multitasking friendly.
while (TRUE)
          {
                    if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
                              {
                                        if (msg.message == WM_QUIT)
            break ;

          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
      else
          {
                                        DoInv() ;
          }
      }

          return (msg.wParam) ;
}

// ---------------------------------- InitApplication ---------------------------------
BOOL InitApplication(HANDLE hInstance)
{
          WNDCLASS wndclass;

          wndclass.style = 0 ;
          wndclass.lpfnWndProc = WndProc ;
          wndclass.cbClsExtra = 0 ;
          wndclass.cbWndExtra = 0 ;
          wndclass.hInstance = hInstance ;
          wndclass.hIcon = LoadIcon (hInstance, szAppName) ;
          wndclass.hCursor = LoadCursor (hInstance, IDC_ARROW) ;
          wndclass.hbrBackground = GetStockObject (BLACK_BRUSH) ;
          wndclass.lpszMenuName = NULL ;
          wndclass.lpszClassName = szAppName ;

          return(RegisterClass (&wndclass));
}

// ----------------------------------  InitInstance ---------------------------------
BOOL InitInstance(HANDLE hInstance, WORD nCmdShow)
{
          hWnd = CreateWindow (szAppName,
                    "Invid",
                    WS_MINIMIZEBOX | WS_POPUP | WS_CAPTION | WS_SYSMENU,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    WINDOW_WIDTH, WINDOW_HEIGHT ,
                    NULL,
                    NULL,
                    hInstance,
                    NULL) ;
```

```
        ShowWindow (hWnd, nCmdShow) ;
        UpdateWindow (hWnd) ;
        return (TRUE);
}

// ------------------------------------ WndProc ------------------------------------
long FAR PASCAL WndProc (HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
{
        int i;

        for(i=0; i < dim(messages); i++)
        {
                if(wMsg == messages[i].Code)
                        return((*messages[i].Fxn)(hWnd, wMsg, wParam, lParam));
        }

        return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}

// ---------------------------------  DoCreate --------------------------------
// DoCreate() is called when our window receives a WM_CREATE message
// at startup.
LONG DoCreate(HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
{
// Call a function called BaseInit that will initialize the variables in the missile base
// structure Base.
        BaseInit();
        return 0;
}

// ---------------------------------  BaseInit --------------------------------
// This function initializes the variables and loads the missile base's
// bitmap.
void BaseInit(void)
{
                Base.x = 40;
                Base.y = WINDOW_HEIGHT - 80;
                Base.Bmp = LoadBitmap(hInst, "base");
}

// ---------------------------------  DoDestroy --------------------------------
// DoDestroy is called when our program receives a WM_QUIT message
LONG DoDestroy(HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
{
// Delete the handle to the missile base's bitmap before exiting. (otherwise Windows
// system resource's won't be released.
        DeleteObject(Base.Bmp);
        PostQuitMessage (0) ;
        return 0 ;
}

// ---------------------------------  DoKeyDown --------------------------------
// DoKeyDown is run whenever our program receives a WM_KEYDOWN message.
```

```
LONG DoKeydown(HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
{
        switch (wParam) {
                case VK_RIGHT :
                        fRightKeyDown = TRUE ;
                        break  ;
                case VK_LEFT :
                        fLeftKeyDown = TRUE ;
                        break  ;
   }
    return 0 ;
}

// ----------------------------------  DoKeyUp ---------------------------------
// DoKeyUp is run whenever our program receives a WM_KEYUP message.
LONG DoKeyUp(HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
{
        switch (wParam) {
                case VK_RIGHT :
                        fRightKeyDown = FALSE ;
                        break  ;
                case VK_LEFT :
                        fLeftKeyDown = FALSE ;
                        break  ;
        }

        return 0 ;
}

// ----------------------------------  DoInv ---------------------------------
// This function runs whenever Windows isn't doing anything else
void DoInv(void)
{
        BaseAnimation (hWnd);
        return ;
}

// ----------------------------------  BaseAnimation ---------------------------------
// This function draws the bitmap of the base at a new position based
// on whether the cursor keys are pressed
void BaseAnimation (void)
{

// Move the x coordinate of the missile base based on which cursor
// key is pressed.
// Base.x += AnimStep is shorthand forBase.x = Base.x + AnimeStep
        if (fRightKeyDown) Base.x += AnimeStep;
        if (fLeftKeyDown) Base.x -= AnimeStep;

// Keep the missile base on the screen
        if (Base.x < 5)  Base.x = 5;
        if (Base.x > (WINDOW_WIDTH-50)) Base.x = WINDOW_WIDTH-50;
```

```
// Another Petzold function slightly modified to make it simpler to use.
// (If you havn't bought Programming Windows by Charles Petzold yet,
// stop reading this and go to your local bookstore and buy it now!)
// The function takes three arguments.  The first argument is a handle to
// the bitmap that will be drawn.  The second and third parameter is the
// x and y position of the bitmap to be drawn.
        DrawBitmap (Base.Bmp, Base.x, Base.y);
}

// ---------------------------- DrawBitmap ----------------------------------
// This is where all the action takes place.  It's great for getting a bitmap
// on your window as simply as possible.  Unfortunately it has performance
// problems when you try and use it for all your graphics work.  (This  is probably
// why Windows doesn't have this function built in)  Because of
// this I'm not going to cover getting a bitmap to the screen  in detail
// until the next issue when we'll start optimizing the drawing of bitmap
// graphics.
void DrawBitmap (HBITMAP hBitmap, int xStart, int yStart)
{
// Declare a handle to a device context.  A handle for a Device Context is needed
// to draw to a window.
        HDC hdc;

// Declare a variable bm of type BITMAP.  It is used to store information about
// a bitmap such as its size and colordepth.
        BITMAP bm ;

// Declare a handle to a device context.  This handle will be to a block of memory
// that will be used as temporary storage for the bitmap that will be drawn.
        HDC hdcMem ;

// Get a handle for a device context for our Window.
        hdc = GetDC(hWnd);

// Get a handle for a device context to a block of memory.  The memory context will
// be based on the handle to the device context of our Window.  That way it will be the
// correct size to store the bitmap.   This saves us from having to calculate the
// size of the bitmap in bytes and allocating a block of memory to hold the bitmap.
        hdcMem = CreateCompatibleDC (hdc) ;

// Put the bitmap into the block of memory that was created above.
        SelectObject (hdcMem, hBitmap) ;

// Get information about the bitmap and put it into the structure bm
        GetObject (hBitmap, sizeof (BITMAP), (LPSTR) &bm) ;

// The BitBlt function copies the bitmap in memory (hdcMem) to the screen (hdc)
        BitBlt (hdc, xStart, yStart, bm.bmWidth, bm.bmHeight,
    hdcMem, 0, 0, SRCCOPY) ;

// Free up the resources we used before leaving
        DeleteDC (hdcMem) ;
        ReleaseDC (hWnd, hdc);
```

}

When you run the program, it will put a small bitmap on the screen that moves back and forth when you press the cursor keys. If you're running it on a 486, you'll notice that it moves too fast to be controlable in a game. That's why we'll need to add our own internal timing to the program to make the animation rate constant.

In the next issue, we'll add the invaders and cover BitBlt'ing in detail. 'Til then
" "

About the author:

(short version)
SWM 25 ISO SWF. Enjoys long walks, sun, surf, sking. Must have own compiler.

(long version)
Bernard Andrys is a engineer at CSI, an ISDN networking development company. He holds a bachelor's degree in Mechanical Engineering from the University of Maryland at College Park. He can be reached through the Internet at andrys@csisdn.com or on the Windows Programming Journal BBS.

(longer version)
I was born in the house my father built. ...No wait, that's some else's biog. umm... Ok... I bet you're wondering what a Mechanical Engineering is doing working for a networking company and writing Windows program. Well I'm wondering too. If you find out, write me. ...and another question. Is it the company I keep or do all programmers like The Hitchhiker's Guide to the Galaxy and Red Dwarf?