

# Introduction to Double Buffering in .NET

## BitBlt

Pronounced “bit-blit”, this is a very important function, which you should know about in spite of the fact that it is *not* directly exposed for use in the Foundation Class Libraries.

*BitBlt* stands for *Bit-Block Transfer*. It means that a “block” of bits, describing a rectangle in an image, is copied in one operation. Usually the graphics card supports this command in hardware, so that the operation can be carried out within the “vertical retrace interval” which we discussed in an earlier lecture. The use of *BitBlt* avoids flicker when displaying an image.

There is a function in the Win32 API of this name, which also occurs in MFC, but the FCL does not provide this function to you directly. Nevertheless it is essentially packaged for use as the *Graphics.DrawImage* method.

## Memory DC:

DC means “device context”. This is represented in the FCL as a *Graphics* object. So far, the *Graphics* objects we have used in our programs usually corresponded to the screen; and in one lecture, we used a *Graphics* object that corresponded to the printer. But it is possible to create a *Graphics* object that does not correspond to a physical device. Instead, it just has an area of RAM (called a buffer) that it writes to instead of writing to video RAM on the graphics card. When you (for example) draw a line or fill a rectangle in this *Graphics* object, nothing changes on the screen (even if you call *Invalidate*), since the memory area being changed by the graphics calls is not actually video RAM and has no connection with the monitor.

This “virtual *Graphics* object” is loosely referred to as a *memory DC*. It is a “device” that exists only in memory; but usually its pixel format does correspond to a physical device such as the screen, so that when data is copied from this buffer to video memory, it is correctly formatted.

## **Double Buffering**

“Double buffering” refers to the technique of writing into a memory DC and then *BitBlt*-ing the memory DC to the screen. This works as follows: your program can take its own sweet time writing to a memory DC, without producing any delay or flicker on the screen. When the picture is finally complete, the program can call *BitBlt* and bang! Suddenly (at the next vertical retrace interval) the entire contents of the memory DC’s buffer are copied to the appropriate part of video RAM, and at the next sweep of the electron gun, the picture appears on the screen. This technique is known as *double buffering*. The name is appropriate because there are two buffers involved: one on the graphics card (video RAM) and one that is not video RAM, and the second one is a “double” of the first in the sense that it has the same pixel format.

[Some books reserve this term for a special case, in which the graphics card has two buffers that are alternately used to refresh the monitor, eliminating the copying phase. But most books use the term *double buffering* for what we have described.]

Whatever is stored in the memory CDC will not be visible, unless and until it gets copied to the CDC that corresponds to the screen. This is done with *BitBlt*, so that the display happens without flicker.

## Why use double buffering?

Double buffering can be used whenever the computations needed to draw the window are time-consuming. Of course, you could always use space to replace time, by storing the results of those computations. That is, in essence, what double-buffering does. The end result of the computations is an array of pixel information telling what colors to paint the pixels. That's what the memory DC stores.

This situation arises all the time in graphics programming. All three-dimensional graphics programs use double-buffering. MathXpert uses it for two-dimensional graphics. We will soon examine a computer graphics program to illustrate the technique.

Another common use of double-buffering is to support animation. If you want to animate an image, you need to "set a timer" and then at regular time intervals, use BitBlt to update the screen to show the image in the next position. The BitBlt will take place during the vertical retrace interval. Meantime, between "ticks" of the timer, the next image is being computed and drawn into the memory DC. In the next lecture, this technique will be illustrated.

**There is a checkbox for double-buffering on my *Form1* property sheet. Can't I just use that and not bother to learn about how to program double buffering?**

For simple applications, such as eliminating annoying flicker, that might work. But you won't get control over *when* the time-consuming computations are done that way. In addition, ten years from now programming tools will have changed two or three times, but the underlying principles of double-buffering will still be the same, so if you learn the fundamentals, it will be useful for your whole professional life, while if all you learn is to check a

certain checkbox in Visual Studio, that probably won't help you ten years from now.

## **When should the memory DC be created?**

Obviously it has to be created when the view is first created. Less obviously, it has to be created again when the view window is resized. But the memory DC is the same size as the screen DC, which is the same size as the client area of the view window. When this changes, you must change your memory DC accordingly.

When you create a new memory DC, you must also destroy the old one, or else memory will soon be used up by all the old memory DC's (provided the evil user plays around resizing his window many times).

It turns out that every window receives a *Resize* event when it is being resized, and it also receives a *Resize* event soon after its original creation (when its size changes from zero by zero to the initial size). Therefore, we add a handler for the *Resize* event and put the code for creating the memory DC in the *Resize* message handler.

## **How to create a memory DC in .NET**

In .NET, the buffer of a memory DC corresponds to the area where a *Bitmap* object stores its pixel information. Here are the essential two lines of code. You would want to call code like this in your *Resize* handler. I put it in a method of its own, so that I can call it at other times when I want the image to be redrawn.

```
private void resetDoubleBuffer()  
{  
    Graphics g = CreateGraphics();  
    m_theBitmap = new Bitmap(this.Width, this.Height, g);  
}
```

```
}
```

Here the member variable *m\_theBitmap*, of type *Bitmap*, is used to store the off-screen image. In the following code an extra line has been added:

```
private void resetDoubleBuffer()  
{  
    Graphics g = CreateGraphics();  
    if(m_theBitmap != null)  
        m_theBitmap.Dispose();  
    m_theBitmap = new Bitmap(this.Width, this.Height,g);  
}
```

While C# has garbage collection, images use a lot of memory, and it's not that difficult to clean up after ourselves. When the user is resizing the window, we may get several of these messages rapidly, and in an animation program, we may get a stream of them under control of a timer, and I worry that if we don't take a little responsibility for our own cleanup, we may suddenly find our animation stopped for garbage collection. That would be sloppy, so let's take care of our own cleanup.

## Drawing on the Memory DC

In our first simple example, we don't even need to get a *Graphics* object. The *Bitmap* class has a *SetPixel* method that allows us to change the color of the pixel at specified coordinates. That's just what we want to do.

Incidentally, the *Graphics* class does *not* have a *SetPixel* method, or any method that enables you to set one pixel at a time. Of course, you could draw a line from a point to itself, I suppose, or fill a rectangle one pixel in height and width, but by omitting *SetPixel*, the designers of the FCL were trying to tell you something: If you need to set one pixel at a time, probably you should be using a *Bitmap* as described in this lecture to do it.

The type of program we're considering here is one that has some method for taking two integers  $i,j$  and computing what color the pixel at  $(i,j)$  should be. Let's say that function is called *colorFunction*, and it returns a *Color*. Then to write on our off-screen buffer, we just need a loop that calls

```
Color c = colorFunction(i,j);  
m_theBitmap.SetPixel(i,j,c);
```

for all the pixels  $(i,j)$  that we want to color. Then when we're finished with that, all we need to do is call *Invalidate*. Our *Paint* handler can look like this:

```
private void Form1_Paint(object sender,  
System.Windows.Forms.PaintEventArgs e)  
{ Graphics g = e.Graphics;  
  g.DrawImage(m_theBitmap,0,0);  
}
```

That leaves unresolved only one major issue: *When* do we do the (possibly lengthy) calculations, i.e. the loop over all pixels  $(i,j)$ ? If the resolution is high and the rectangle is large, there can easily be millions of pixels, and if each pixel requires a thousand calculations, that could take seconds, even on a fast machine in 2005. If we're trying to do an animation in which we need to update 30 times a second, we are pushing the machine's capabilities.

## **Performing the computations during otherwise idle cycles**

When do we want to draw in the memory DC? Bear in mind that this is a time-consuming operation because we have to carry out a non-trivial computation *for each pixel*.

So, we don't want to wait for a *Paint* event, because it can take a long time to do the computations. Instead, we want to be

computing anytime nothing else is happening. The key to arranging this is the `WM_IDLE` message, which Windows sends our application whenever the application message queue is otherwise empty. In Visual Studio, you won't see the *Idle* event listed as a possible event to handle in your form. That's because it is the *Application* that has a message queue, not each window owned by the application.

I didn't find a way to add an *Idle* handler through the form editor, but it's only one line of code. To show where it goes, I've exhibited the entire *Form1* constructor:

```
public Form1()  
{  
    InitializeComponent();  
    Application.Idle += new EventHandler(Form1_Idle);  
    ResizeRedraw = true;  
}
```

Now, once we call *Form1\_Idle*, we have to wait for it to finish. So, we don't want to draw the entire picture before we let Windows back in to process other messages. What if the user is frantically trying to exit the program, and it won't respond until it finishes slowly drawing the picture? Therefore, we draw only *some* columns of the picture at a time. I will show how that is done. Suppose we have a method

```
void DrawMandelbrot(Bitmap b, int FirstColumn, int k)
```

that draws *k* columns of the picture, starting with *FirstColumn*. To write *Form1\_Idle*, we introduce member variables

```
private bool m_NewDrawing = true;  
private int m_CurrentColumn = 0;
```

The former will be set to *false* when we finish with the drawing, and set to *true* again when the user makes data changes that require

a new drawing. Here is a suitable *Form1\_Idle*. The variable *k* controls how many columns are drawn during the processing of a single *Idle* event.

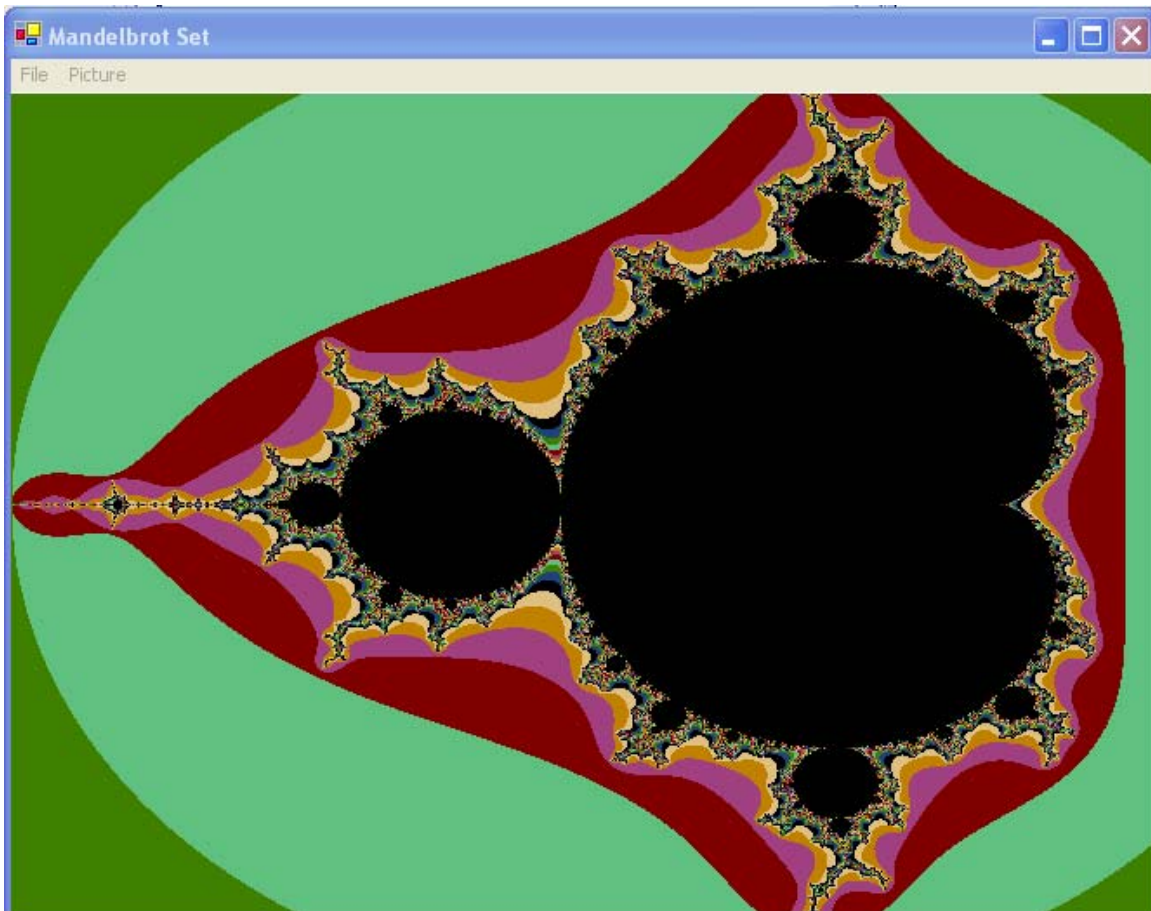
```
private void Form1_Idle(object sender, System.EventArgs e)
{
    if(!m_NewDrawing)
        return;
    int k = 1;    // Try 1,20,200, ClientRectangle.Width to see
                 // what it does
    int ncolumns;
    if(m_CurrentColumn + k >= ClientRectangle.Width)
        ncolumns = ClientRectangle.Width-m_CurrentColumn;
    else
        ncolumns = k;
    /* Draw ncolumns columns of pixels of the drawing */
    DrawMandelbrot(m_theBitmap,m_CurrentColumn,ncolumns);
    Rectangle r = new Rectangle(m_CurrentColumn,
                                0,
                                ncolumns,
                                ClientRectangle.Height);
    m_CurrentColumn +=ncolumns;
    if(m_CurrentColumn >= ClientRectangle.Width)
    { m_CurrentColumn = 0;
      m_NewDrawing = false;
    }
    Invalidate(r);
}
```

## The Mandelbrot Set

So far, everything applies in complete generality, to any program that computes a picture pixel-by-pixel. There are plenty of mathematical formulas which produce interesting computer graphics that way. Most of them start by associating the screen rectangle on which we will draw with a rectangle in the  $x$ - $y$  plane. Then each pixel corresponds to coordinates  $x,y$ , and we will make some computation  $count(x,y,m\_ct)$ , where  $m\_ct$  is an arbitrary number (say 200) that is used to stop a computation that might otherwise go on indefinitely. The result of this computation is an integer value. We then use an arbitrary color scheme to assign a

color to the pixel based on this integer. The association of colors to integers is completely up to us--we get to exercise our artistic creativity.

The Mandelbrot set was one of the first such sets to attract widespread interest. It was discovered by B. Mandelbrot of IBM Research. Here's a screen shot of the program that is the example in this lecture:



In computing the Mandelbrot set, the screen rectangle on which we will draw is associated with some rectangle in the complex number plane. Each pixel is thus assigned a complex number, say  $c$ . We then define  $f(z) = z^2 + c$ . Define

$$z[0] = c, \text{ and}$$

$$z[n+1] = f(z[n]).$$

We compute these numbers  $z[n]$  for  $n = 1, 2, \dots, 200$ . If at some point we find  $|z[n]| > 2$ , we stop, and color the pixel according to the integer  $n$ . If we make it to  $n=200$ , we stop and color the pixel black.

```
int count(double ax, double ay, int ct)
/* ax and ay are the real coordinates of a pixel;
   ct is the iteration limit. Here is where we do the
   only computation specific to the Mandelbrot set.
   Return the number of iterations.
*/
{
  int i=0;
  double x,y,xy,xsq,ysq;
  x=xsq=y=ysq=0.0;
  for(i=0;i<ct && xsq+ysq<4;i++)
  {
    xsq=x*x;
    ysq=y*y;
    xy=x*y;
    x=xsq-ysq+ax;
    y=2*xy+ay;
  }
  return(i);
}
```

## Other Details of the Mandelbrot Program

The program also involves the following details:

- Conversion between integer (device, or pixel) coordinates and floating-point coordinates representing a point in the complex plane.
- Allowing the user to select a new view rectangle by dragging the mouse (as in our first demo of dragging).
- Selecting a color corresponding to the integer computed for each pixel.

None of these points is central to the theme of the lecture, namely double buffering.

Further features you might want to implement:

- A dialog that lets the user select a new rectangle by specifying its complex coordinates. There is a menu item to bring up this dialog, but it is not implemented in the posted version of the program.
- A zoom feature: use the arrow keys to zoom in or out by a factor of 2.
- “panning”, dragging the picture sideways with the right mouse button. Processor speeds are perhaps able to support this feature now.