EE 356
WriteableBitmap Notes

In WPF you do not write graphics to the screen. Instead, you create a *Visual* and a *DrawingContext*, render it to a bitmap, and set the bitmap as the source of an *image*. This has the advantage that it frees the programmer from writing paint events for graphics. It can, however, be slow and restricting for things such as automata which needs to write to a very small area of the screen many times over. The WriteableBitmap class addresses this problem. To use the WriteableBitmap you need to be `using System.Windows.Media.Imaging;`

Normally, in WPF, bitmaps, like strings, are immutable. Once you create one you can't change it – you can create another and delete the original but you can't modify the existing bitmap. Writeable bitmaps, as the name implies, can be changed and WPF has a WriteableBitmap class. You can create a WriteableBitmap, change its pixels on the fly, and assign it as the source of an image.

A WriteableBitmap is typically declared like this:

```
Writeablebitmap wbmap =
   new WriteableBitmap(100, 100, 300, 300, PixelFormats.Bgra32, null);
```

The first two parameters are the size in pixels – in this case 100 x 100. The second two parameters are the resolution which is 300 dots/inch x 300 dot/inch. The pixel format is Bgra32 which means each pixel has a byte value for the blue, green, red, and alpha values. (Alpha values specify the transparency). The null parameter which is last allows you to specify a palette if the format needs one. The Bgra32 pixel format needs one 32-bit word for each pixel. There are a large number of pixel formats but this one is simple enough and suitable for what we do in this class.

We would like to be able to read or write a pixel or a group of pixels at address x, y on the bitmap. Memory addresses are linear – not two dimensional and the bitmap data is stored in this linear address space and the WriteableBitmap structure does not know the length of the rows and columns that you used to create the bitmap. The bitmap data is stored as rows with each row following the last. If the row length is *r* bytes and we want the pixel at *x, y* the linear address will be $x + y*r$. There are complications! For some pixel formats the number of bits per pixel is not 32 or even a power of two so it is possible, for example to have a row of pixels that is not *n* bytes long where *n* is an integer. Further, Windows forces each new row to begin on a byte address divisible by 4. If the (bits/pixel)/8 times the number of pixels in a row is not divisible by four, zeros are added as padding at the end of the row to force the next row to begin on an address divisible by 4. The row length in pixels plus the padding length is called the *stride*. The *stride* is the actual amount of storage needed to store a single row. You only need to be concerned about the stride if you are trying to address a writeable bitmap using an x-y address scheme (as we are).

Back Buffer

The WriteableBitmap class has a separate buffer called the *back buffer*. You can write to the back buffer, declare the back buffer to be *dirty*, and when it is time to refresh the screen the back buffer will be copied to the screen. The back buffer property returns a pointer to the back buffer which allows you to access it. Unfortunately, when you use pointers in C# you create unsafe

code since the pointer values are addresses whose value can be data dependent. Your program could therefore access unauthorized memory making it unsafe. There is a fix for this which I will discuss later.

The article at:
http://www.i-programmer.info/programming/wpf-workings/527-writeablebitmap.html
presents much of the information which I have presented here and gives two methods which can be used to address a WriteableBitmap using an x, y addressing scheme. The two methods are SetPixel and GetPixel. Listings are given on the following pages. Using these two methods you can set or get the color of an individual pixel at address x, y. Notice that both methods have code blocks marked as unsafe. If you put these methods into your code you will have to change the compiler options for your project to get it to compile. To do so, right click on the project name and select *Properties*. Under properties select the *Build* tab. Check the box that says "Allow unsafe code".

The code below uses SetPixel to color a 100 x 100 writeable bitmap all white: The color alpha value must be 255 to be opaque and 0 to be transparent.

```
WriteableBitmap wbmap = new WriteableBitmap(100, 100,
                    300, 300, PixelFormats.Bgra32, null);
for(x=0;x<100;x++)
  {for(y=0;y<100;y++)
     {//Make everything white
      SetPixel(wbmap, x, y,Color.FromArgb(255, 255, 255, 255));
     }
  }
```

To read a pixel using GetPixel at location 50, 50 use the following:

```
Color c = new Color();
c = GetPixel(wbmap, 50, 50);
```

You can then extract the red, green, blue, and alpha values using c.R, c.G, c.B, and c.A.

```
public void SetPixel(WriteableBitmap wbm, int x, int y, Color c)
   {if (y > wbm.PixelHeight - 1 || x > wbm.PixelWidth - 1) return;
    if (y < 0 || x < 0) return;
    if (!wbm.Format.Equals(PixelFormats.Bgra32))return;
    wbm.Lock();
    IntPtr buff = wbm.BackBuffer;
    int Stride = wbm.BackBufferStride;
    unsafe
      {byte* pbuff = (byte*)buff.ToPointer();
       int loc=y *Stride  + x*4;
       pbuff[ loc]=c.B;
       pbuff[loc+1]=c.G;
       pbuff[loc+2]=c.R;
       pbuff[loc+3]=c.A;
      }
    wbm.AddDirtyRect(new Int32Rect(x,y,1,1));
    wbm.Unlock();
   }

public Color GetPixel(WriteableBitmap wbm, int x, int y)
   {if (y > wbm.PixelHeight - 1 || x > wbm.PixelWidth - 1)
       return Color.FromArgb(0, 0, 0, 0);
    if (y < 0 || x < 0)
       return Color.FromArgb(0, 0, 0, 0);
    if (!wbm.Format.Equals(PixelFormats.Bgra32))
       return Color.FromArgb(0, 0, 0, 0);;
    IntPtr buff = wbm.BackBuffer;
    int Stride = wbm.BackBufferStride;
    Color c;
    unsafe
      {byte* pbuff = (byte*)buff.ToPointer();
       int loc = y * Stride + x * 4;
       c=Color.FromArgb(pbuff[loc+3], pbuff[loc+2],
                        pbuff[loc+1], pbuff[loc]);
      }
    return c;
   }
```