Image Manipulation in Windows Mobile 5

Capturing and Manipulating Photographic Images in Windows Mobile 5 Devices

Rob Miles

Department of Computer Science, University of Hull

August 2007

Applies to:

Windows Mobile™ 5 based devices
Microsoft® .NET Compact Framework version 2.0
Microsoft Visual Studio® .NET 2005

Summary: Using the camera control and image loading dialog to capture images for use in games. Performing image processing to convert the images into sprites for use in mobile games. Performing image processing to generate game backgrounds.

There are a number of downloads at various stages in the development.

Download Sample Image Capture

Download Full Image Processing Game

Contents

Introduction
What You Will Need to Get Started
The Camera in a Mobile Device
Getting Started with Visual Studio 2005 and Mobile Devices
The Microsoft.WindowsMobile.Forms Resource
Using the CameraCaptureDialog Form
Using the SelectPictureDialog Form
Sample Image Application
Image Processing to create Sprites
Working with Bitmaps
Speeding up Bitmap Access
Unsafe Code
Moving through the Bitmaps
Controlling the Edit Process
Managing the Cursor Bitmap
Saving Bitmap data into image files
Creating the Background
Color Manipulation
Image Processing
The Wild Color Filter
The Noise Filter
Selecting Image Processing Actions
Sample Project
Future Work

©2007 OpenNETCF Consulting, LLC
For more information visit http://community.OpenNETCF.com

Introduction

Windows Mobile 5.0 offers the prospect of games based on images which the player captures with the onboard camera. In this article you will discover how to capture such images and prepare them for use as sprites and backgrounds in a game.

What You Will Need to Get Started

You will need Visual Studio .NET 2005. You will also need to acquire the Windows Mobile 5.0 SDK for the device that you are targeting. This is a free download.

You don't actually need a phone handset or mobile device because you will be using techniques that make it possible to do all the work on the emulators provided by Microsoft Visual Studio® .NET 2005. However, the emulators do not support the camera behavior, although you can use the image loading dialog.

If you have a device, it will make things much more interesting. Only the most recent Windows Mobile 5 devices have Compact Framework Version 2.0 built in, but this will be installed automatically as required when the program is downloaded into the device.

If you want to download the programs into a real device you will also need Active Sync version 4.1. The programs have been tested on an SPV C500 and an Imate Jasjar.

The Camera in a Mobile Device

Whilst Smartphones have always been supplied with cameras, and many Pocket PC devices now sport them, it is only with Windows Mobile 5.0 that programmers are able to use them easily. In previous devices the camera has been controlled by software which was specifically written by the hardware manufacturers. This software did not provide a way in which programmers writing in managed code could easily interact with the camera to initiate the taking of pictures. That has changed in the latest version of Windows Mobile, it is now possible for a program to initiate the taking of photographs by the device owner.

However, it is important to understand just what level of use you can make of the camera. There is no direct camera control from a C# program, in that you cannot order the camera to take a picture. Instead your program displays a camera capture dialog form. When the user has taken a picture control is returned to your program.

If you want direct control of the camera device itself this can be achieved by use of the DirectShow interface which is also supported by Windows Mobile 5.0. However, for the purpose of simple games, allowing the user to take a photograph is sufficient.

Getting Started with Visual Studio 2005 and Mobile Devices

The kind of project that you need is a Visual Studio 2005 .NET Compact Framework Version 2.0 forms application.

©2007 OpenNETCF Consulting, LLC
For more information visit http://community.OpenNETCF.com

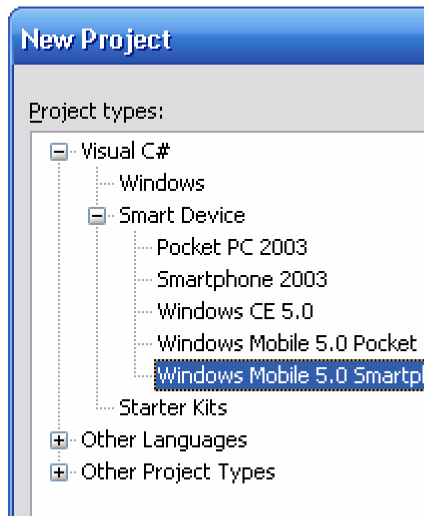


Figure 1. Creating the new project

Note that you only get the Windows Mobile 5.0 projects once you have installed the appropriate software development kits (SDK). The examples that are going to be created will target the Windows Mobile 5.0 Smartphone with a QVGA display. All the code can be used on other platforms with only minimal changes.

Once you have created a new project, the next thing to consider is the form factor of the target device. At present Smartphone devices are available with two screen resolutions, the 176x220 of the original and 320x240 of more recent devices. The higher resolution is often referred to as QVGA because the screen is quarter the size of a 640x40 VGA screen. You can select the screen resolution by using the `FormFactor` property of the form:

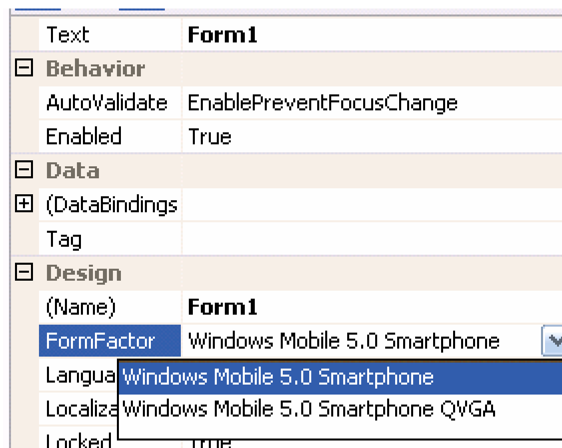


Figure 2. Selecting the FormFactor

Note that as a general rule you should make sure that your program will work with either size screen. However this feature is useful for allowing you to match your program with a particular sized device. I am going to be using the camera on device with QVGA resolution, so I have selected this option. You should select the option which matches your target device.

The Microsoft.WindowsMobile.Forms Resource

The `CameraCaptureDialog` is actually supplied in the `Microsoft.WindowsMobile.Forms` library. This is part of the Windows Mobile 5.0 SDK, but is not automatically added as a reference when a project is built. To add the library, right click on the References item in the Solution Explorer, select Add Reference from the menu which appears and then add the library:

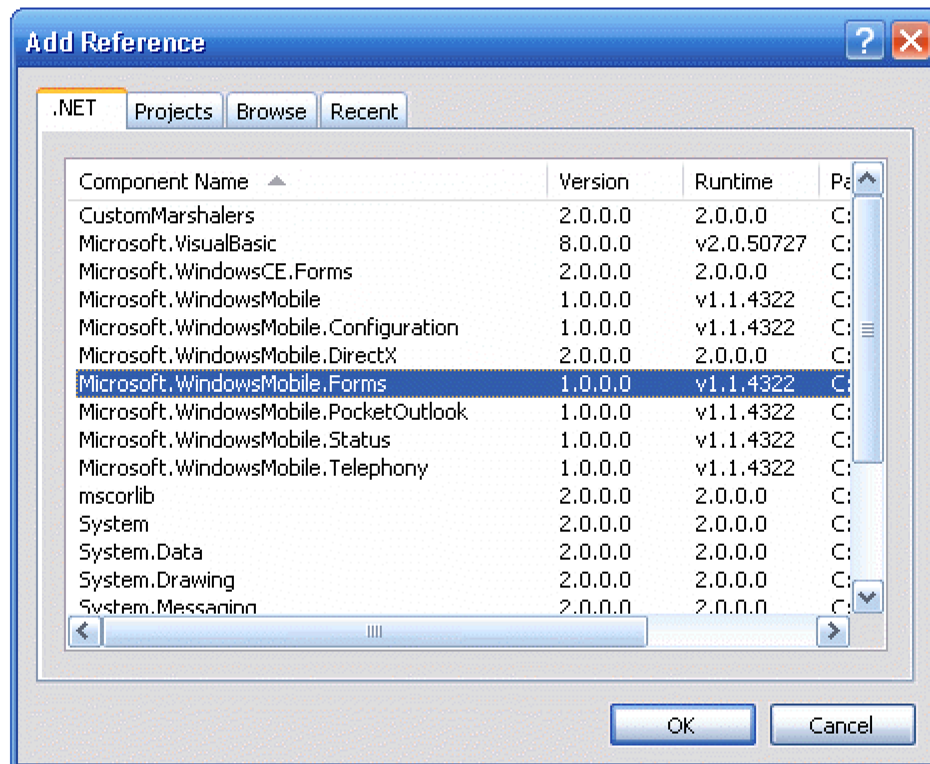


Figure 3. The WindowsMobile Forms library

Now our program can create and use all the items in this library, including the `CameraCaptureDialog`.

Using the CameraCaptureDialog Form

A camera capture dialog is created like any other dialog box, and also shown in the same way. The code to take a picture is as follows:

```
CameraCaptureDialog cameraDialog = new CameraCaptureDialog();
if ( cameraDialog.ShowDialog()==DialogResult.OK ) {
    // if we get here the picture has been taken correctly
    // the FileName property gives the name of the image file
}
```

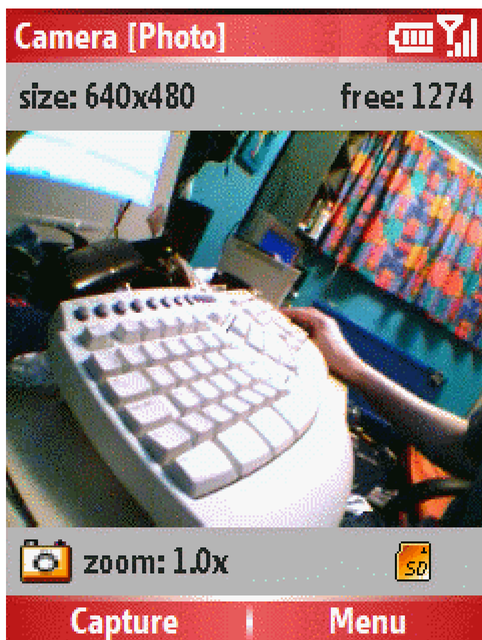
Note that the dialog does not return an image; instead it functions like a file dialog, in that it returns the name of the file where the image has been stored. If you want to use the image in your program you must then load the image from the file. The camera

capture dialog has a cancel option, so your program must always allow for the action being cancelled.

As with other windows forms, it is important to restore the original form once the picture has been taken. The method below will use the `CameraCaptureDialog` to take a picture and return the name of the picture file if this succeeds. It is called from within an existing form, and will return the display to that form when it finishes.

```
private string takePicture()
{
    string result = null;
    CameraCaptureDialog cameraDialog = new CameraCaptureDialog();
    cameraDialog.Mode = CameraCaptureMode.Still;
    cameraDialog.StillQuality = CameraCaptureStillQuality.High;
    cameraDialog.Resolution = new Size(640, 480);
    if ( cameraDialog.ShowDialog()==DialogResult.OK ) {
        result = cameraDialog.FileName;
    }
    this.Show();
    cameraDialog.Dispose();
    return result;
}
```

Note that it the code also sets properties of the dialog to select a high quality still image at a resolution of 640x480 (although the user can override these when the picture is taken). When the method runs the camera capture dialog on the Smartphone is displayed:



©2007 OpenNETCF Consulting, LLC

Figure 4. A Camera Capture Dialog

Note that the precise appearance of this dialog, and the options which are available for the camera, will vary from one Windows Mobile device to another.

Using the SelectPictureDialog Form

Sometimes the user may wish to select an existing picture from a file instead. Windows Mobile 5 provides an additional dialog which can be used to select a picture. It is used in exactly the same way as the camera dialog.

```
SelectPictureDialog selectDialogue = new SelectPictureDialog();


if (selectDialogue.ShowDialog() == DialogResult.OK)
{
    // if we get here the picture file has been selected
    // the FileName property gives the name of the image file
}
```

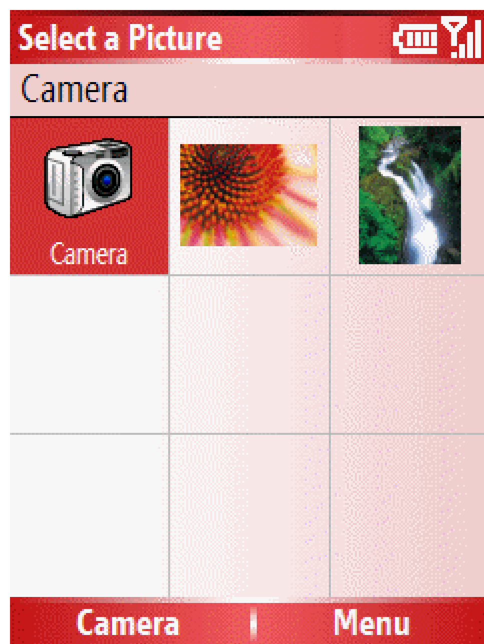


Figure 5. A Select Picture Dialog

The `SelectPictureDialog` also contains the option of taking a picture with the camera as well. If the user does this, the name of the file containing the picture that was taken is returned.

Sample Image Application

The sample application allows you to load an image or take a picture using the camera. The image which is loaded is displayed in a picture box.

©2007 OpenNETCF Consulting, LLC
For more information visit http://community.OpenNETCF.com

Image Processing to create Sprites

Capturing full sized pictures is a start, but if they are to be used in games they need to be converted into images which can be used as the basis of game sprites. This means that they will probably need to be cropped down, so that only part of the picture is to be used. As a plain edged sprite will not be very interesting, it would also be useful to be able to create sprites with a softer edge, and with more interesting shapes.

This can be achieved by the use of a mask image which is combined with part of the captured image to produce shapes.

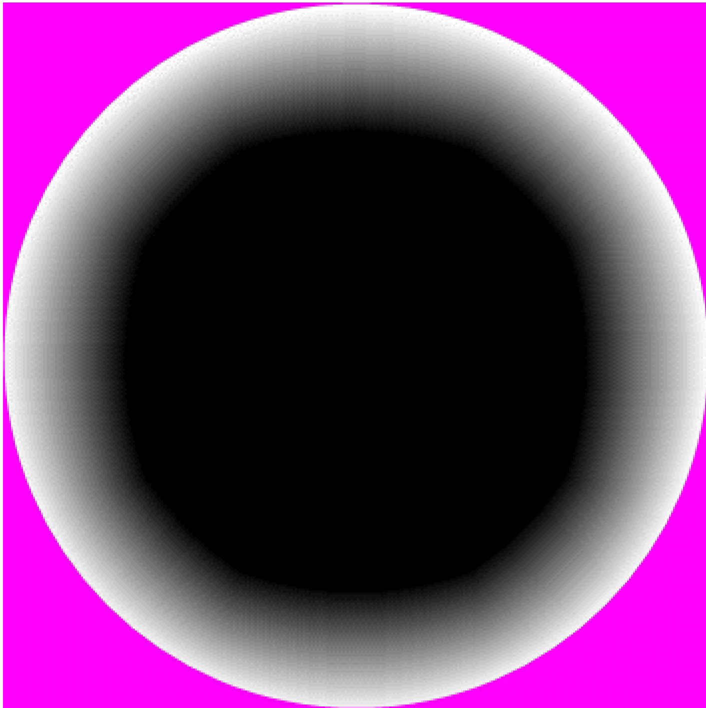


Figure 6. A Round Mask

The figure shows a mask which can be used to create a round sprite. The purple background color will be set as the transparent color when the sprite is drawn. The other colors on the mask will be added to the existing pixels so that the sprite appears to get lighter towards the edge. The center of the circle is black, so that part will have no effect on the image. Our clipping code must resize the image and add the mask to get the desired effect.

Figure 7. Applying the Mask

©2007 OpenNETCF Consulting, LLC

Figure 7 shows the mask after it has been applied to part of a picture. The sprite that is produced has been resized to an appropriate resolution and then drawn on a black background.

Working with Bitmaps

It is comparatively easy to get hold of the color of a particular pixel in an image. The `Bitmap` class provides a method called `GetPixel` which returns the color at a particular pixel position in the bitmap:

```
Color pixelCol = myBitmap.GetPixel(0,0);
```

This would obtain the color of the pixel in the top left hand corner of the bitmap and set the variable `pixelCol` to that value. There is a corresponding `SetPixel` method which can be used to set the color of a pixel:

```
myBitmap.SetPixel(0,0,Color.Red);
```

This would set the color of the pixel in the top left hand corner of the bitmap to red.

The sprite generation program must combine pixels in the mask with those in the source image. Pixels in the source image which are in the same position as transparent ones in the mask must be set to the transparent color, otherwise the mask pixel color values are added to the source ones:

```
Color maskColor = mask.GetPixel((int)maskX, (int)maskY);
if (maskColor.Equals(transparentColor))
{
    source.SetPixel(sourceX, sourceY, transparentColor);
}
else {
    if (!maskColor.Equals(Color.Black) ) {
        Color sourceColor = source.GetPixel(sourceX, sourceY);
        int red = masterColor.R + maskColor.R;
        if (red > 255) red = 255;
        int green = masterColor.G + maskColor.G;
        if (green > 255) green = 255;
        int blue = masterColor.B + maskColor.B;
        if (blue > 255) blue = 255;
        source.SetPixel(sourceX, sourceY,
            Color.FromArgb( red, green, blue));
    }
}
```

This code fetches a pixel from the mask at location `maskX`, `maskY`. If the mask pixel is the same color as the transparent color the source image pixel at `sourceX`,`sourceY` is

set to transparent. If the pixel is not the transparent color the program adds the color of the pixel in the mask is added to the color of the source.

Note that the color value is limited to the range 0-255, to stop strange effects if the value wraps around. If the above statements are performed on each pixel on the image, and the X and Y values are updated appropriately, this will perform the image processing that is required. Unfortunately it takes a long time to do this. On a standard Smartphone it can take a few minutes to process even a small image.

Speeding up Bitmap Access

The problem is that the `GetPixel` and `SetPixel` methods are very slow. The managed C# is trying to access the low level bitmap information and this takes time. To improve the speed the program must access this low level information directly. It turns out that this is quite easy. The `Bitmap` class provides a method called `LockBits` which will provide a pointer to the bitmap image data in memory. It also locks this data in position, so that it will not be moved around by the system.

```
BitmapData bitmapBase = myBitmap.LockBits(
   bounds,                           // bounding rectangle on the bitmap
   ImageLockMode.ReadWrite,          // mode of the access
   PixelFormat.Format24bppRgb) ;     // required format – 8 bits per color
```

The actual image data is presented as a sequence of pixel values for each row of the image in turn. In other words an image which is 60 pixels wide and 40 pixels high would have 180 bytes of data for the top row, followed by 180 bytes for the next, and so on.

The `BitmapData` provides us with the base of the screen, as a pointer to bytes:

```
private Byte* bitmapBaseByte;

bitmapBaseByte = (Byte*) bitmapBase.Scan0.ToPointer();
```

The pointer variable `bitmapBaseByte` now points at the beginning of the raw data for our image. In other words, since first byte of a pixel is the blue component for that pixel, the code:

```
*bitmapBaseByte = 0xFF;
```

- would se the blue component of the top left hand pixel on the screen to maximum.

The actual pixel data is placed in memory as three consecutive 8 bit values for blue, green and red respectively. The best way to manipulate this in C# is to use a structure:

```
public struct PixelData
{
    public byte blue;
    public byte green;
    public byte red;
}
```

By careful use of casting the program can pull out the color components in each pixel:

```
PixelData * currentPixel = *((PixelData*)( bitmapBaseByte));
```

©2007 OpenNETCF Consulting, LLC
For more information visit http://community.OpenNETCF.com

```
currentPixel->green = 0;
```

This would set the green component of the top left hand pixel to minimum.

The program can move down the image to the next pixel by incrementing the pointer. To move to the next row of the image the pointer must be increased by the size of the row. There is a slight complication in that the number of bytes in each row is must always be a multiple of four when the bitmap data is mapped into memory. If this is not the case then the end of the row is padded out with "blank" bytes up to the nearest four byte boundary.

In the case of a row which contains 50 pixels the actual number of bytes in each row would be 152, i.e. 50 pixels multiplied by 3 bytes per pixel gives 150. This is then lifted up to 152, that being the nearest value to 150 which is divisible by 4. This makes the calculation of the width of each row slightly more complicated:

```
byteWidth = pixelWidth * 3;


if (byteWidth % 4 != 0)
{
    byteWidth += (4 - (byteWidth % 4));
}
```

The `pixelWidth` value is multiplied by three (blue, green and red bytes) to get the `byteWidth` value. If this width is not divisible by four it is adjusted upwards by the appropriate amount.

Once the raw data manipulation is complete the bitmap must be unlocked so that it can be returned to normal use:

```
myBitmap.UnlockBits(bitmapBase);
```

Unsafe Code

Note that this means that our C# code is no longer safe, in that we are following pointers directly into memory, which bypasses all the usual safety checks performed in managed code. This means that the class which contains methods which do this must be declared as `unsafe`:

```
public unsafe class ProcessBitmaps {
```

For this to build correctly, the project must be modified to allow unsafe code to be compiled. This setting can be found on the Build tab in the Project Properties:

©2007 OpenNETCF Consulting, LLC
For more information visit http://community.OpenNETCF.com

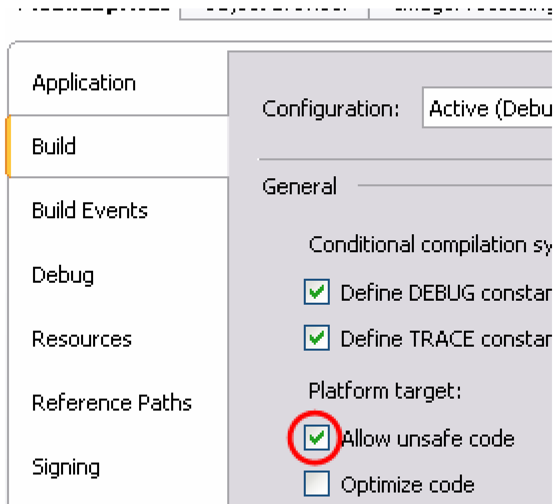


Figure 8. Allowing unsafe code in the project

Figure 8 shows how this option is selected. If the program goes wrong, for example it tries to access a byte outside the range of the bitmap, it will not actually crash the host device, but it will fail without any kind of error report or exception being thrown.

Moving through the Bitmaps

When the sprite is being built the program must move through the source bitmap and combine the pixels in it with pixels in the mask bitmap. However, the bitmaps will not necessarily be the same shape and size. This means that the program must map the pixels from one bitmap into the matching ones in the other. This can be achieved by making the step through the mask bitmap the same relative size as the step through the image bitmap:

```
float maskXStep =
   (float)maskBitmap.Width / (float) sourceRectangle.Width;
float maskYStep =
   (float)maskBitmap.Height / (float) sourceRectangle.Height;
float maskX = 0;
float maskY = 0;
```

The `maskBitmap` is the `Bitmap` containing the mask image. The variable `sourceRectangle` is a rectangle which describes the region of the source rectangle which is to be masked. The user selects this during the edit process. These two floating point steps are then used to update the position in the mask bitmap as each pixel in the source is processed. The step value corresponds to the movement of a single pixel in the given direction.

```
for (int y = sourceRectangle.Top; y < sourceRectangle.Bottom; y++)
{
    pPixel = (PixelData*)(sourcepBase + y * sourceWidth +
        sourceRectangle.Left * sizeof(PixelData));
```

©2007 OpenNETCF Consulting, LLC

```
    mPixel = (PixelData*)(maskpBase + (int)maskY * maskWidth);
    for (int x = sourceRectangle.Left; x < sourceRectangle.Right; x++)
    {
        if (mPixel->red == transparentPixel.red &&
            mPixel->green == transparentPixel.green &&
            mPixel->blue == transparentPixel.blue)
        {
            // if the mask pixel color is the trasparent mask,
            // overwrite the image pixel with our transparent colour
            pPixel->red = transparentPixel.red;
            pPixel->blue = transparentPixel.blue;
            pPixel->green = transparentPixel.green;
        }
        else
        {
            // add the mask colour onto the image colour
            value = mPixel->red + pPixel->red;
            if (value > 255) value = 255;
            pPixel->red = (byte)value;
            value = mPixel->green + pPixel->green;
            if (value > 255) value = 255;
            pPixel->green = (byte)value;
            value = mPixel->blue + pPixel->blue;
            if (value > 255) value = 255;
            pPixel->blue = (byte)value;
        }
        int oldMaskX = (int)maskX;
        maskX += maskXStep;
        int dist = ((int)maskX) - oldMaskX;
        mPixel += dist;
        pPixel++;
    }
    maskX = 0;
    maskY += maskYStep;
}
```

©2007 OpenNETCF Consulting, LLC
For more information visit http://community.OpenNETCF.com

The code above performs the masking process. The pixels are taken from a rectangle in the source image and combined with the appropriate ones in the mask image. If the source pixel corresponds with one which is set to the transparent color in the mask, the source pixel is set to transparent as well.

Controlling the Edit Process

The idea behind the program is that the user will take a picture with the camera and then select it for use as a sprite in the game. To do this they will have to move a cursor around the screen and also change the size of the cursor to fit the required region.

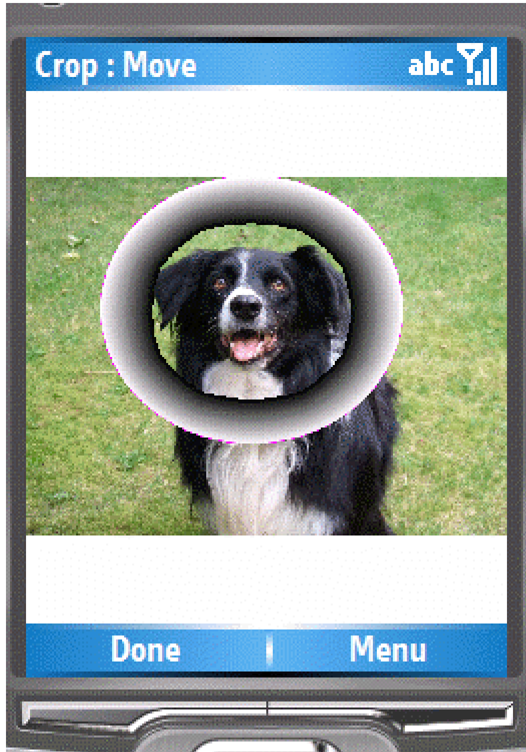


Figure 9. Selecting the part of the image to clip

The cursor can be made to operate in two modes, either moving over the screen or changing in size. The two modes are toggled by pressing in on the joystick. When the required part of the image has been selected the use presses the Done menu key to complete the image processing and create the sprite itself. The menu is used to select the cursor mode and also to allow the user to choose between a range of differently shaped masks for the mask cursor.

Managing the Cursor Bitmap

The first version of the program was written so that the actual image processing was performed during the image selection, in other words the mask was applied in real time so that the user was able to see the exact result of the crop action. Unfortunately, the processor in the Smartphone is not sufficiently powerful to do this and provide an acceptable user interface, so instead the mask was converted into a cursor image which is overlaid on the background.

This image is obtained by converting all the black portion of the mask image into a transparent color, and then drawing the result on top of the background. A method, `MakeBlackTransparent`, was created which performs this simple image processing task.

When the user selects a new mask another image is created which is used in this way.

Saving Bitmap data into image files

Windows Mobile 5 provides a very simple mechanism for saving bitmaps into files in a variety of image formats. The `Image` class provides a `Save` method which is supplied with the path to the file to be created and the image format:

```
myBitmap.Save(@"\My Documents\My Pictures\Bitmap.bmp", ImageFormat.Bmp);
```

This would save the image in `myBitmap` into a file `Bitmap.bmp` with in the windows bitmap format. The `ImageFormat` enumerated type also lets you specify GIF, JPEG and PNG formats for the save operation. The path above causes the picture to be saved in the `My Pictures` subdirectory of the `My Documents` folder.

Creating the Background

The program can now produce interestingly shaped sprites which can then be used in a game. However, we also need a background for the sprites to move over. This can also be based on an image which was taken with a built in camera, but the image processing will be quite different.

Rather than crop portions out of an image we will instead be performing image processing on the picture to make it more appropriate as a background. To do this the user may want to make the image brighter or darker, apply a color tint or add noise or other effects. The fundamental behavior of the code will be very similar, in that it will involve manipulating the colors in image data, but the action will be quite different:

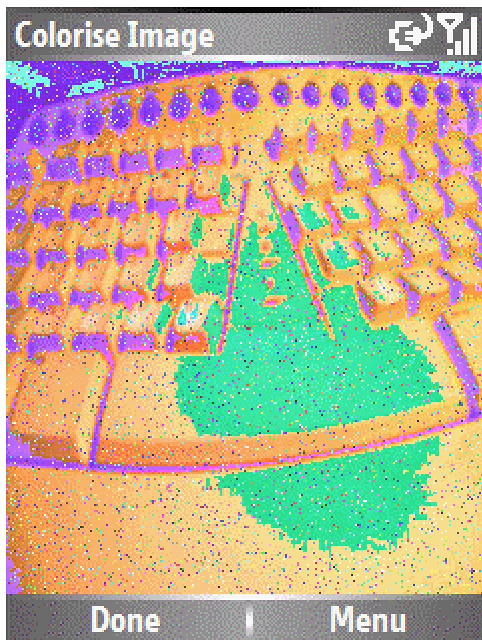


Figure 10. An Image Processed Keyboard

©2007 OpenNETCF Consulting, LLC
For more information visit http://community.OpenNETCF.com

The above image shows a fairly boring picture of a keyboard which has been given the "wild" color process, had some noise added and then been increased in brightness to make an interesting background.

Color Manipulation

When performing image processing only the source image is required. There is no masking involved. The color manipulation can be performed by simply adding an adjustment value to each of the red, green and blue components of a pixel. The best way to do this is to make a single method which accepts the change values for each of the components. This single method can then be used tint an image or change the brightness simply by passing different data values in. The actual processing is quite simple, but the program must make sure that the color values do not go beyond the allowed 8 bit values:

```
PixelData* pPixel;
for (int y = 0; y < source.Height; y++)
{
    pPixel = (PixelData*)(sourcepBase + y * sourceWidth);
    for (int x = 0; x < source.Width; x++)
    {
        int redVal = pPixel->red + redChange;
        if ( redVal <0 ) redVal = 0;
        if ( redVal > 255) redVal = 255;
        pPixel->red = (byte)redVal;
        int greenVal = pPixel->green + greenChange;
        if ( greenVal <0 ) greenVal = 0;
        if ( greenVal > 255) greenVal = 255;
        pPixel->green = (byte)greenVal;
        int blueVal = pPixel->blue + blueChange;
        if (blueVal < 0) blueVal = 0;
        if (blueVal > 255) blueVal = 255;
        pPixel->blue = (byte)blueVal;
        pPixel++;
    }
}
```

This code applies each of the change values, which can be either positive or negative, to perform the tinting action.

©2007 OpenNETCF Consulting, LLC
For more information visit http://community.OpenNETCF.com

Image Processing

Most image processing programs provide a range of effects which can be used to make a picture more interesting. These include actions such as adding drawing effects and adding noise or remapping the color palette. We are going to use a couple of simple filters which will provide the user with some scope to make the background more artistic. These filters will operate only on single pixels, for effects such as blurring or lens distortion the code gets more complex as values must be transferred from one pixel to another.

The Wild Color Filter

One way to get a "wild" color effect is to add a random offset to the components in the color of a pixel. This will shift the color balance in potentially interesting ways. In this case the program does not limit the maximum values of the components, but instead allows them to wrap around the limits to get more interesting effects:

```
int redVal = (pPixel->red + redChange) % 256;

pPixel->red = (byte)redVal;

int greenVal = (pPixel->green + greenChange) % 256;

pPixel->green = (byte)greenVal;

int blueVal = (pPixel->blue + blueChange) % 256;

pPixel->blue = (byte)blueVal;
```

In the code above the values of `redChange`, `greenChange` and `blueChange` are picked at random for the entire pass through the image. This very simple technique seems to result in genuinely interesting and often very artistic looking changes to the image. By using a fixed seed for the random number generator it is possible to make a given set of changes reproducible each time the program is used.

The Noise Filter

The noise filter randomly replaces pixels in the image with ones which have colors which have been randomly chosen. The amount of noise which is added can be controlled by the range of random numbers which trigger the swap action. A figure of 5% noise seems to give appropriate levels of noise:

```
for (int y = 0; y < source.Height; y++)

{

    pPixel = (PixelData*)(sourcepBase + y * sourceWidth);

    for (int x = 0; x < source.Width; x++)

    {

        if (rnd.Next(100) < 5)

        {

            pPixel->red = (byte)rnd.Next(255);

            pPixel->green = (byte)rnd.Next(255);

            pPixel->blue = (byte)rnd.Next(255);
```

©2007 OpenNETCF Consulting, LLC

```
        }
        pPixel++;
    }
}
```

Selecting Image Processing Actions

Each of the image effects is selected from a given menu option. However, if the user wants to repeat a given action a number of times this would become tedious. The program has therefore been constructed so that it tracks the most recently used image processing option and then repeats this when the joystick is pushed in. This memory is managed in terms of a delegate instance which is set to point to the most recently used image processing method.

```
delegate void imageProcessItem () ;
```

The delegate `imageProcessItem` can be used to create delegate instances which refer to methods which are void and accept no parameters. The class contains a delegate which keeps track of the most recently used image processing method:

```
private imageProcessItem doAddNoiseProcessItem = null;
```

This is initially set to `null`, but when a method is called the delegate is made to refer to that method:

```
private imageProcessItem doBlueTintItem = null;
private void doBlueTint()
{
    Cursor.Current = Cursors.WaitCursor;
    ProcessBitmaps.ProcessColor(previewImage, 0, 0, 10);
    if (doBlueTintItem == null)
    {
        doBlueTintItem = new imageProcessItem(doBlueTint);
    }
    currentItem = doBlueTintItem;
    Invalidate();
    Cursor.Current = Cursors.Default;
}
```

Rather than create a new delegate instance each time a method is called, the program instead creates a delegate for each method which is then assigned as appropriate. The code above is called when the user selects the "Blue Tint" option. First it sets the cursor to wait. Next it calls the `ProcessColor` method – passing it the image and the red, green and blue change value. Then it creates the delegate if required and sets the current item. Finally it puts the cursor back.

Sample Project

The sample project contains all the above behaviors. A single sprite can be cropped created and will then be moved over a background image, which can be selected and processed as required. The main class, `GameForm`, implements the movement of the sprites over the background. The `MakeSpriteForm` class loads in images and provides the user interface for the sprite mask selection and cropping process. The `MakeBackgroundForm` class loads in images and provides the user interface for the image processing actions. All the actual image processing is provided by the `ProcessBitmaps` class, which provides a number of static methods that perform the appropriate image updates.

Future Work

The program at the moment does the image capture and processing, but it does not include any of the game behavior. It is also not possible to perform any image processing on the sprites themselves. Adding these features, and making a game out of the supplied sprites would not be difficult, and is left as an exercise for the reader.

©2007 OpenNETCF Consulting, LLC
For more information visit http://community.OpenNETCF.com