Beating the System: Fun With Bitmaps

ScanLine secrets revealed!

by Dave Jewell

Recently I had occasion to review a certain popular graphics toolkit. The core functionality of this product is implemented as a set of DLLs, callable from Visual Basic, Visual C++, Delphi, C++ Builder or whatever development environment takes your fancy. For Delphi and C++Builder, the vendors also supply a set of VCL components that effectively 'wrapped' the various DLL calls, making it relatively easy to perform sophisticated bitmap manipulations via drop-in components.

Although the underlying DLLs were quite good, I wasn't so impressed with the quality of the VCL controls and, while deliberating on a suitable subject for this month's article, it occurred to me that it might be fun to see what sort of bitmap manipulations could be performed using Delphi itself. After all, why bother with third-party OCX files and DLLs if you can do the job yourself from within a single EXE file?

The Pixels Array: Just Say No!

In order to perform bitmap manipulations, we need to find a mechanism for getting at the individual pixels themselves.

If you've ever delved around inside the GRAPHICS.PAS source file (and if you haven't, you should!) then you may well have discovered the deeply weird `Pixels` property of the `TCanvas` object. Object Pascal is quite a simple language when compared to C++ but, as I'm fond of saying, it offers a surprisingly rich set of syntactic constructs. A good example of this is the support for array properties, of which the `Pixels` array is an excellent example. Here is the declaration of this property:

```
property Pixels[X, Y: Integer]:
  TColor read GetPixel
  write SetPixel;
```

If you've never used array properties in your components then you should try them out. In the case of the `Pixels` property, it allows you to access individual pixels of the canvas just as if they were a two-dimensional array. The compiler converts read and write references to the `Pixels` property into calls on `GetPixel` and `SetPixel`, both of which are `private` methods. Thus, you can do something like this:

```
Canvas.Pixels[0,0] := clBlack;
```

In terms of generated code, this will get transmogrified into a call on the `SetPixel` method, with the `X` and `Y` co-ordinates as the first two parameters and `clBlack` as the third. This approach is elegant and reasonably efficient... until it hits Windows! Internally, the `GetPixel` and `SetPixel` routines map straight down onto the Windows API routines of the same name and ultimately end up calling code within the installed video device driver. The bottom line is that these two routines (and therefore the `Pixels` property itself) are very expensive in terms of execution time. Typically, even the simplest bitmap manipulation requires that the code read and write every single pixel in the image, and you can therefore see that the `Pixels` property represents a horrendously slow way of getting the job done. Is there a better way? Indeed there is...

Enter The ScanLine Property

A far superior solution is to make use of the `ScanLine` property. The `Pixels` array is a property of the `TCanvas` class, but `ScanLine` is a property of `TBitmap`. `ScanLine` is implemented as a one-dimensional array property, accessed by row or Y co-ordinate. For each row of pixels in the bitmap, `ScanLine` will give you a direct pointer to the raw bitmap data. This means that we can manipulate the bitmap data a row at a time, and we only ever have to call the VCL library when we want the next row of data. Even then, the underlying `GetScanLine` method is very efficient and essentially does little more than recalculate an offset into the bitmap data for a given row.

That said, there is one potential problem: the actual format of pixel data returned by `ScanLine` will vary depending on the format of the bitmap itself. If you look at the help documentation for `TBitmap`, you'll notice that there's another property called `PixelFormat`. Using this property, it's possible to specify the format that you want to work with. Subject to certain restrictions, you can load a bitmap into a `TBitmap` object and then set the `PixelFormat` to massage the image data into the format that you want to work with. The Borland documentation doesn't go into a great deal of detail about what the different pixel formats are, so I've briefly described them for you in Table 1, over the page.

For the sake of simplicity, all the bitmap-twiddling code developed here is based around the `pf32bit` format. These days, it's reasonable to assume that any 32-bit Delphi developer will be equipped with a system capable of displaying 24-bit colour, and any new image processing code should certainly be written to that standard. Moreover, the `pf32bit` pixel format offers the highest possible level of performance, and that's what we're after! In passing, it's worth noting that pretty much the *only* advantage of the aforementioned `Pixels` property array is its ability to manipulate any bitmap in a format-independent manner: all the dirty work of figuring out and accessing the underlying pixel format is left to the GDI and device driver code. As with Visual Basic, simplicity is not always synonymous with performance.

Brightening Up Your Bitmaps!

Armed with the new ScanLine property, let's see how easy it is to manipulate a bitmap. The code in Listing 1 implements a new TGraphic descendant of TBitmap called TExBitmap. It performs essentially the same job as TBitmap, but with a number of extra bells and whistles.

To begin with, you'll see that I've defined a new scalar type called TExBrightness in the range -255 through to +255. This type is used to specify the brightness of a bitmap. The reason that it varies from -255 to +255 is simple: as I've already indicated, we're basing this code around the pf32bit pixel format which means that the colour value of any pixel can vary between 0 and 255. For example, if the red, green and blue components of a particular pixel are all zero, then of course it's a black pixel. In order to crank the brightness of this black pixel all the way up from black to brilliant white, we need to be able to add up to 255 to each colour value. Similarly, if we're starting with a white pixel, and we want to fade it down to black, then we may need to subtract as much as 255 from each colour channel. Therefore, by defining a data type in the range -255 to +255, we can add this number to the colour value of each pixel in a bitmap and thereby alter its brightness.

However, I'm jumping ahead of myself. Back in Listing 1, you'll see a set of type declarations just after the implementation part of the unit. These types deliberately override (and therefore hide) the type declarations of the same name that are defined in the standard Delphi GRAPHICS.PAS and WINDOWS.PAS files. Ordinarily, this would be a rather risqué thing to do, but since it's inside a unit's private implementation, it isn't going to screw up anyone else. There are two reasons why I've done this. Firstly, the Microsoft/Borland supplied version of TRGBQuad defines the three colour value fields as rgbBlue, rgbGreen and rgbRed. Being terminally lazy, and knowing that these fields are going to be repeatedly referenced within the bitmap manipulation code, I chose to replace them with the more snappily-named r, g, b fields.

As an aside, if you ever discover an erroneous declaration in a Borland or third-party unit, this is a good technique for fixing the declaration without having to alter someone else's code. And if you want to access the original declaration after overriding it, you can still do so by qualifying the type name with the name of the containing unit, for example Windows.TRGBQuad. Isn't Delphi Pascal wonderful?

Format	Explanation
pfDevice	This pixel data format is only used for device-dependent data.
pf1bit	One bit per pixel format. For obvious reasons, this format is only relevant to black and white images, or image masks. The pixel data is simply stored as consecutive bits, with eight pixels of information per byte.
pf4bit	As the name suggests, this format corresponds to 4 bits per pixel. Since 2^4=16, this gives a maximum of 16 possible colours. Each byte of the ScanLine data stores two pixels of information. Furthermore, this data format uses palettes, the pixel values (0..15) are simply indexes into the palette information where the real colours are stored.
pf8bit	This is a variation on the above. This time, 256 colours are possible because a full byte is used to store each pixel. As before, the pixel values index into a palette.
pf15bit	A rather odd data format in which each pixel occupies a 16-bit word. The most significant bit of the word is zero. This is followed by five bits of red colour information, 5 bits of green, and 5 of blue. In other words, the intensities of the three primary colours can each take one of 32 discrete values. A total of 32,768 different colours can be displayed using this data format, which isn't palette based.
pf16bit	Similar to the previous format, but six bits are allocated to the green component, meaning that the most significant bit of the word is now used, the red component having to 'shuffle up' by one bit. The theory here is that the human eye has increased sensitivity to green, so it makes sense to allow twice as many levels of green as for red and blue. 65,536 different colours can be represented.
pf24bit	Now we get to the serious stuff! With this data format, each pixel is represented by three consecutive bytes (red, green, blue order) which specify up to 256 different values for each colour. 2^{24} = 16,777,216, meaning that over 16 million different colours are possible in 24-bit mode, greater than the human eye's ability to distinguish.
pf32bit	This pixel format is indistinguishable from the preceding format, except that a full 32-bits are required to store each pixel. As before, one byte is allocated for each of the three colours, with an additional, spare byte also allocated.

➤ *Table 1*

Now here's another, more serious reason why I chose to redefine the TRGBQuadArray type. As you can see from the original declaration of this routine inside GRAPHICS.PAS, this type is intended to correspond to the pixel data that's returned from a call to ScanLine when the PixelFormat is set to pf32bit. Unfortunately, whoever wrote this type definition apparently figured that no bitmap would ever be wider than 256 pixels, and the Byte type was therefore used to specify the range of the array. I've simply replaced this with a Word, as shown below, to prevent any objections from the compiler

```
unit EXBitmap;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Dialogs;
type
  TExBrightness = -255..255;
  TExBitmap = class (TBitmap)
  private
    fChangeLock: Boolean;  // True if we're altering image
    fOriginal: TBitmap;    // original bitmap image
    fBrightness: TExBrightness; // current brightness level
    fFlipped: Boolean;     // if image is vertically flipped
    fMirrored: Boolean;    // if image horizontally mirrored
    fInverted: Boolean;    // if image inverted (negative)
    fBlurRadius: Double;   // radius for Gaussian blur
    fWeights: array [-100..100] of single;
    fSize: 1..100;
    procedure SetBrightness (Value: TExBrightness);
    procedure SetMirrored (Value: Boolean);
    procedure SetInverted (Value: Boolean);
    procedure SetFlipped (Value: Boolean);
    procedure SetBlurRadius (Value: Double);
    procedure InitGaussianWeightings;
    procedure BlurRow (S, D: Pointer; Count: Integer);
  protected
    procedure Changed (Sender: TObject); override;
  public
    constructor Create; override;
    destructor Destroy; override;
    procedure GaussianBlur;
    property Flipped: Boolean read fFlipped
      write SetFlipped default False;
    property Mirrored: Boolean read fMirrored
      write SetMirrored default False;
    property Brightness: TExBrightness read fBrightness
      write SetBrightness default 0;
    property Inverted: Boolean read fInverted
      write SetInverted default False;
    property BlurRadius: Double read fBlurRadius
      write SetBlurRadius;
  end;

implementation
type
  // Override the definitions in Windows.pas
  PRGBQuad = ^TRGBQuad;
  TRGBQuad = packed record
    r: Byte;
    g: Byte;
    b: Byte;
    rgbReserved: Byte;
  end;
  // Override the definitions in Graphics.pas
  TRGBQuadArray = array [Word] of TRGBQuad;
  PRGBQuadArray = ^TRGBQuadArray;

constructor TExBitmap.Create;
begin
  Inherited Create;
  fBlurRadius := 3.0;
  fOriginal := TBitmap.Create;
end;

destructor TExBitmap.Destroy;
begin
  fOriginal.Free;
  Inherited Destroy;
end;

procedure TExBitmap.Changed (Sender: TObject);
begin
  Inherited Changed (Sender);
  if not fChangeLock then begin
    PixelFormat := pf32Bit;
    fBrightness := 0;
    fFlipped := False;
    fMirrored := False;
    fInverted := False;
    // Force a *COPY* of the bitmap.  >>DON'T<< call Assign!
    fOriginal.Width := Width;  fOriginal.Height := Height;
    fOriginal.Canvas.Draw (0, 0, Self);
    fOriginal.PixelFormat := pf32Bit;
  end;
end;

procedure TExBitmap.SetBrightness (Value: TExBrightness);
var
  Row, Col: Integer;
  Line: PRGBQuadArray;
begin
  if (not Empty) and (fBrightness <> Value) then begin
    fBrightness := Value;
    fChangeLock := True;
    // Get an unadulterated copy of the image
    Canvas.Draw (0, 0, fOriginal);
    Assert (PixelFormat = pf32Bit);
    for Row := 0 to Height - 1 do begin
      Line := ScanLine [Row];
      for Col := 0 to Width - 1 do
        with Line [Col] do
          if Value > 0 then begin
            if r + Value > 255 then r := 255
              else Inc(r, Value);
            if g + Value > 255 then g := 255
              else Inc(g, Value);
            if b + Value > 255 then b := 255
              else Inc(b, Value);
          end else begin
            if r + Value < 0 then r := 0
              else Inc(r, Value);
            if g + Value < 0 then g := 0
              else Inc(g, Value);
            if b + Value < 0 then b := 0
              else Inc(b, Value);
          end;
      end;
      fChangeLock := False;
      Inherited Changed (Self);
  end;
end;

procedure TExBitmap.SetFlipped (Value: Boolean);

  procedure FlipBitmap (bmp: TBitmap);
  var
    TempScanLine: Pointer;
    ScanLineBytes, Row, H: Integer;
  begin
    with bmp do begin
      Assert (PixelFormat = pf32Bit);
      H := Height;
      ScanLineBytes := Width * sizeof (TRGBQuad);
      GetMem (TempScanLine, ScanLineBytes);
      for Row := 0 to ((H and (-2)) - 1) div 2 do begin
        Move (ScanLine [Row]^, TempScanLine^,
          ScanLineBytes);
        Move (ScanLine [H - Row - 1]^, ScanLine [Row]^,
          ScanLineBytes);
        Move (TempScanLine^, ScanLine [H - Row - 1]^,
          ScanLineBytes);
      end;
    end;
    FreeMem (TempScanLine);
  end;
begin
  if (not Empty) and (Value <> fFlipped) then begin
    fFlipped := Value;
    FlipBitmap (Self);
    // Lossless operation - so apply to original also.
    FlipBitmap (fOriginal);
    Inherited Changed (Self);
  end;
end;

procedure TExBitmap.SetMirrored (Value: Boolean);

  procedure MirrorBitmap (bmp: TBitmap);
  var
    Temp: TRGBQuad;
    Row, Col, W: Integer;
    Line: PRGBQuadArray;
  begin
    with bmp do begin
      Assert (PixelFormat = pf32Bit);
      W := Width;
      for Row := 0 to Height - 1 do begin
        Line := ScanLine [Row];
        for Col := 0 to ((W and (-2)) - 1) div 2 do begin
          Temp := Line [Col];
          Line [Col] := Line [W - Col - 1];
          Line [W - Col - 1] := Temp;
        end;
      end;
    end;
  end;
begin
  if (not Empty) and (Value <> fMirrored) then begin
    fMirrored := Value;
    MirrorBitmap (Self);
    // Lossless operation - so apply to original also.
    MirrorBitmap (fOriginal);
    Inherited Changed (Self);
  end;
end;

procedure TExBitmap.SetInverted (Value: Boolean);
  procedure InvertBitmap (bmp: TBitmap);
  var
    Row, Col: Integer;
    Line: PRGBQuadArray;
  begin
    with bmp do begin
      Assert (PixelFormat = pf32Bit);
      for Row := 0 to Height - 1 do begin
        Line := ScanLine [Row];
        for Col := 0 to Width - 1 do
          with Line [Col] do begin
            r := not r;
            g := not g;
            b := not b;
          end;
        end;
    end;
  end;
begin
  if (not Empty) and (Value <> fInverted) then begin
    fInverted := Value;
    InvertBitmap (Self);
    // Lossless operation - so apply to original also.
    InvertBitmap (fOriginal);
    Inherited Changed (Self);
  end;
end;
procedure TExBitmap.InitGaussianWeightings;
const
  delta: Double = 1.0 / 510;  // Smaller entries are ignored
  LastRadius: Double = 0.0;
var
  Idx: Integer;
  D: Double;
  procedure Normalise (Lo, Hi: Integer);
  var
    Total: Double;
    Idx: Integer;
  begin
    Total := 0;
    for Idx := Lo to Hi do
      Total := Total + fWeights[Idx];
    for Idx := Lo to Hi do
      fWeights [Idx] := fWeights[Idx] / Total;
  end;
begin
  // If same radius as requested last time, nothing to do...
  if fBlurRadius = LastRadius then Exit;
  LastRadius := fBlurRadius;
  //Init weights array with standard deviation = fBlurRadius
  for Idx := Low (fWeights) to High (fWeights) do begin
    D := Idx / fBlurRadius;
    fWeights [Idx] := exp (-D*D/2);
  end;
  // Normalise around maximum bounds
  Normalise (Low (fWeights), High (fWeights));
  // Discard entries smaller than Delta
  fSize := High (fWeights);  D := 0;
  while (D < delta) and (fSize > 1) do begin
    D := D + 2 * fWeights [fSize];
    Dec (fSize);
  end;
  // Normalise again, using new bounds
  Normalise (-fSize, fSize);
end;
procedure TExBitmap.BlurRow (S, D: Pointer; Count: Integer);
var
  Idx, Pix, j, n: Integer;
  rr, gg, bb, w: Double;
  Src: PRGBQuadArray absolute S;
  Dest: PRGBQuadArray absolute D;
begin
  for j := 0 to Count - 1 do begin
    rr := 0; gg := 0; bb := 0;
    for n := -fSize to fSize do begin
      w := fWeights [n];  Idx := j - n;
      // Ensure index is pinned between 0..Count-1
      if Idx < 0 then Idx := 0 else if Idx > Count - 1 then
        Idx := Count - 1;
      with Src [Idx] do begin
        rr := rr + w * r;
        gg := gg + w * g;
        bb := bb + w * b;
      end;
    end;
    with Dest [j] do begin
      Pix := Trunc (rr);
      if Pix < 0 then Pix := 0
        else if Pix > 255 then Pix := 255;
      r := Pix;
      Pix := Trunc (gg);
      if Pix < 0 then Pix := 0
        else if Pix > 255 then Pix := 255;
      g := Pix;
      Pix := Trunc (bb);
      if Pix < 0 then Pix := 0
        else if Pix > 255 then Pix := 255;
      b := Pix;
    end;
  end;
end;
procedure TExBitmap.SetBlurRadius (Value: Double);
begin
  if Value > 0.0 then fBlurRadius := Value;
end;
procedure TExBitmap.GaussianBlur;
type
  PPRows = ^TPRows;
  TPRows = array[Word] of PRGBQuadArray;
Var
  Rows: PPRows;
  Column, Scratch: PRGBQuadArray;
  ScanLineBytes, H, W, Row, Col: Integer;
begin
  if not Empty then begin
    fChangeLock := True;
    // Get a copy of the original image
    Canvas.Draw (0, 0, fOriginal);
    Assert (PixelFormat = pf32Bit);
    H := Height;  W := Width;
    ScanLineBytes := W * sizeof (TRGBQuad);
    InitGaussianWeightings;
    GetMem (Rows, H * sizeof (Pointer));
    GetMem (Column, H * sizeof (TRGBQuad));
    // Retrieve the address of each bitmap scanline
    for Row := 0 to H - 1 do Rows [Row]:= Scanline [Row];
    // Blur each row
    GetMem (Scratch, ScanLineBytes);
    for Row := 0 to H - 1 do begin
      BlurRow (Rows [Row], Scratch, W);
      Move (Scratch^, Rows [Row]^, W * sizeof (TRGBQuad));
    end;
    // Blur each column
    ReallocMem (Scratch, H * sizeof (TRGBQuad));
    for Col := 0 to W - 1 do begin
      // first read the column into a TRow
      for Row := 0 to H - 1 do Column [Row] := Rows
        [Row][Col];
      BlurRow (Column, Scratch, H);
      // Replace the column in the destination bitmap
      for Row := 0 to H - 1 do Rows [Row][Col] := Scratch
        [Row];
    end;
    FreeMem (Rows);
    FreeMem (Column);
    FreeMem (Scratch);
    fChangeLock := False;
    Inherited Changed (Self);
  end;
end;
end.
```

➤ *Listing 1*

and/or Delphi's runtime range-checking code.

```
 TRGBQuadArray =
   array[Word] of TRGBQuad;
```

Right then, let's look at how one would use the ScanLine facility to vary the brightness of a bitmap. As you can see from Listing 1, TExBitmap exports a Brightness property of type TExBrightness. The 'business end' of this code is the private SetBrightness method that gets called whenever the property is altered. Having checked that the desired brightness level differs from the current brightness, the code first retrieves a copy of the original bitmap (I'll be explaining this later; for now, just concentrate on the way in which pixels are manipulated through the ScanLine property).

Because TExBitmap is derived from TBitmap, we've got instant access to our own properties such as Width, Height and so on. Accordingly, the code is organised with an outer-level FOR loop that iterates through all the available rows in the bitmap, using Height to get the bitmap size. For each row of bitmap pixels, we use ScanLine to retrieve a pointer, Line, to the pixel data. In this case, you'll notice that Line is of type PRGBQuadArray, using our new, improved type declaration. A second FOR loop then iterates through each horizontal pixel in the current scan line, either boosting or reducing the colour values of each pixel, depending on whether or not the Value parameter is positive or negative. The important thing here is to ensure that each colour value is 'pinned' between 0 and 255, rather than simply allowing 250+20 (for example) to 'overflow' into a new colour value of 14.

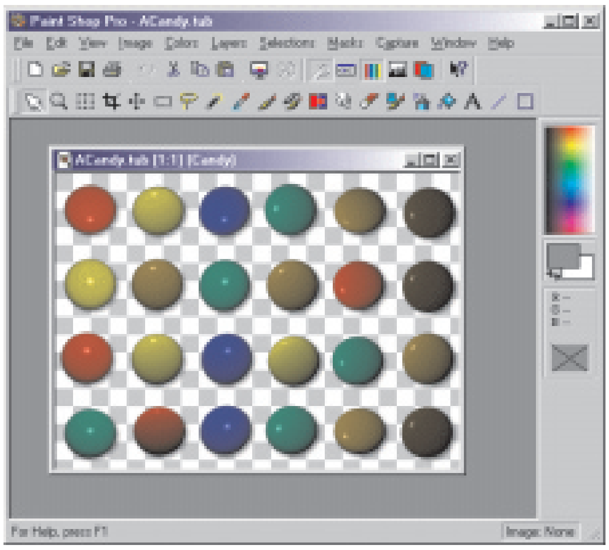
Although this code isn't optimal, it's fast enough to allow us to adjust the brightness of a large bitmap in real-time, as you'll see when we come to look at the test-bed application, later in this article. If you really want to burn rubber, you could resort to tricks such as (for example) pre-calculating 255-Value, and then comparing colour values against this when `Value >= 0`. If you insist upon ultimate performance then you're very welcome to rewrite the whole thing in assembler!

When Information Gets Lost...

Being able to increase or decrease the brightness of a bitmap is great fun, but there is a fundamental difficulty here. Once a colour value has been increased to 255 or 0, any further increase or decrease (respectively) in the brightness of the bitmap will lose information from the image. To put this another way, suppose you've got a pixel which has a red value of 255, a green value of 255 and a blue value of zero. In other words, the pixel is solid yellow. As you increase the brightness of the bitmap on which this pixel resides, the blue colour value will climb steadily towards 255, but the red and green values will stay where they are, having already reached maximum. Thus, information has been lost: the *relative difference* between the colour values within the pixel has been destroyed. To put it simply, the pixel will forget its yellowness. Once you reach pure white (all values equal to 255) reducing the brightness of the image will simply give you successive shades of grey.

What it really boils down to is this: altering the brightness of a bitmap is a non-reversible operation. The chances are that, even when you only reduce the brightness slightly, some pixel colour values will hit zero, while a slight increase in brightness will 'pin' other colour values to 255. This raises serious problems if we want to (for example) create an image processing application that displays a trackbar, allowing the end-user to select the degree of brightness for an image. How do we allow arbitrary changes in brightness, while retaining all the information that the bitmap contains?

The answer, of course, is to create a backup copy of the bitmap. That's the purpose of the private `fOriginal` bitmap contained within the `TExBitmap` class. As you can see from the code, this



➤ *Figure 1: I can't promise to give you all the functionality of PhotoShop or Paint Shop Pro (I wish!) but you will be surprised how easy it is to implement powerful graphics routines in Object Pascal.*

bitmap is automatically constructed when the `TExBitmap` constructor is called, and destroyed when an instance of `TExBitmap` is freed. More importantly, the `TExBitmap` class overrides the `Changed` event handler, ensuring that `TExBitmap.Changed` gets called every time that the image is modified. Thus, whenever a new bitmap is loaded into the object (as, for example, through a call to `LoadFromFile`) this code ensures that a copy of the original image is immediately made, and salted away in the `fOriginal` bitmap.

It would be tempting here to simply use `Assign` to copy the initial bitmap image into `fOriginal`. However, if you've done much work with `TBitmap`, you'll appreciate that this won't work. In the case of the `TBitmap` class, `Assign` is essentially a 'shallow copy' (to use the conventional OOP terminology) which will leave `fOriginal` pointing at the same API-level bitmap image as the enclosing `TExBitmap` object. To put it another way, `TBitmap` has been specifically written so that multiple objects of this class can all refer to the same bitmap, using reference counting to manage destruction of the underlying bitmap data. In an ordinary application, this behaviour is a major benefit (especially when working with large bitmaps) but here, it's essential that any alterations to the `TExBitmap` image don't affect our original copy. Similarly, we can't just copy the `Handle` property of the `TExBitmap` class to the `Handle` of `fOriginal`, it would have the same effect as `Assign`. Instead, I specifically set the size of `fOriginal` to agree with the new bitmap and then draw the image onto the canvas of the bitmap cache, effectively forcing the creation of a completely new, independent bitmap.

You'll also notice that, at the same time, the various properties of `TExBitmap` are reassigned their default values and that the `PixelFormat` property of both bitmap images is set to `pf32Bit`. OK, so far so good, but what's the purpose of this mysterious `fChangeLock` variable, I hear you cry? We need this change lock because, without it, we might inadvertently make some change to the bitmap that would invoke the `Changed` method, thereby overwriting our original bitmap. By setting and clearing this flag within bitmap manipulation routines, we ensure that this can't happen. If you carefully examine the code inside GRAPHICS.PAS, you'll see that `TBitmap.Changed` can be called in a variety of circumstances.

It's Flippin' Marvellous...!

At this point, we've got a nice re-usable class for brightening or darkening a bitmap. But there's lots more that can be done. Back in Listing 1, you'll see that the `TExBitmap` class also has a `Flipped` property. If you set this to True, the bitmap image will automatically be flipped vertically. This works by calling a private method called `SetFlipped`. Within

SetFlipped, the code checks to see if the bitmap is empty and, if not, calls the nested FlipBitmap routine once for the bitmap belonging to the TExBitmap instance, and once for the internal fOriginal image.

So what's the rationale here? Well, the most important point to bear in mind is that flipping a bitmap is a lossless operation. Unlike the earlier discussion on changing the brightness of an image, flipping a bitmap will never reduce the information content. Consequently, it's quite acceptable to flip the original image as well as the 'main image' (for want of a better word). What we can't do is flip the image once, and then copy the flipped image into the original. If we did that, any brightness changes made in the main image would be copied into the original image, making it impossible to subsequently restore the original brightness levels. You'll also notice that this routine does not bother to set the fChangeLock variable because none of its code triggers a call to Changed.

The real meat of the routine is, of course, the FlipBitmap call. Internally, this allocates a temporary memory buffer large enough to store one scanline of pixel information and it then iterates through the bitmap, using the temporary buffer to swap successive scanlines in the image. I've used the Move library routine in order to make things go as fast as possible. If you find yourself scratching your head over that interesting-looking FOR loop, here's an explanation of what's happening: if you think of the image as being made up of an even number of scanlines (say, six, for the sake of argument) then we obviously want to swap scanlines 0 and 5, scanlines 1 and 4, and scanlines 2 and 3. But what if there were seven scanlines in the image? In this case, we'd want to swap the outermost six scanlines but leave scanline four alone. The bottom line is that the scanline swapping code needs to deliberately ignore the centre scanline of any bitmap that's an odd number of pixels in height. By ANDing the height with $FFFFFFFE (-2, in other words) that's the effect we achieve.

In a similar vein, another easy transformation we can apply is mirroring. This is the same as flipping, except that we're reversing the bitmap horizontally, rather than vertically. To accomplish this, I added another property, Mirrored, to the TExBitmap class. As before, the real work is done inside a private method called SetMirrored. Again, this is a lossless operation and so the internal logic of the code is identical to that used by the SetFlipped routine, calling the internal MirrorBitmap routine on both the original and the main images. You'll notice that, this time I'm simply taking each scanline at a time, and swapping pixels using a temporary pixel variable of type TRGBQuad. As with the vertical flipping code, you'll see that the innermost FOR loop takes care to ignore the centre pixel in scanlines which are an odd number of pixels wide: there's no point wasting time swapping a pixel with itself!

As far as easy-peasy transformations are concerned, the simplest possible must surely be the process of inverting an image. Once again, this is handled by a Boolean property, Inverted, and a corresponding 'action procedure' called SetInverted. This is also a lossless transformation which is applied to both the main image and the original. You'll notice that we simply negate each of the three colour values within each pixel and that's it, job done. If you're an assembler aficionado in search of ultimate performance, you can probably do the whole thing using a main loop no more complex than that shown in Listing 2.

```
$1:   xor      dword ptr [ebx], 0FFFFFFFFh       ; invert pixel
      add      ebx,4                              ; step to next pixel
      loop     $1                                 ; loop until done
```

➤ *Listing 2*

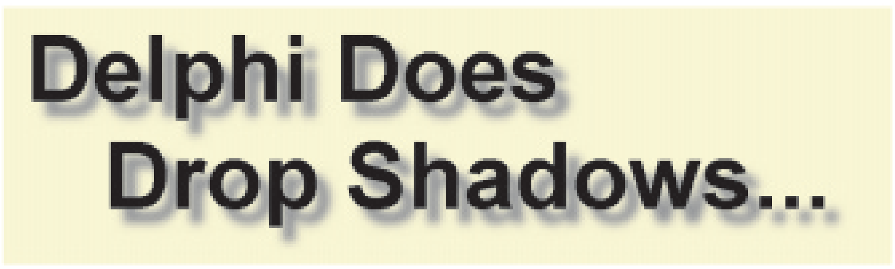
This assembler language fragment is a good demonstration of why the pf32Bit pixel format offers the highest possible level of performance. Stepping through and modifying three bytes at a time would be messier and slower. You'll also notice that in this case, the code happily tramples all over the rgbReserved byte in each TRGBQuad entry. This might cause you to raise an eyebrow, but it will have no ill effects whatsoever.

Gaussian Smoothing: Delphi Meets PhotoShop!

Well, maybe not quite. I wouldn't dare suggest that you can use the routines presented here as a replacement for all the wonderful effects available in PhotoShop, but I hope that this article will persuade you how easy it is to perform image processing using Delphi.

It's a fact of life that some people buy packages like PhotoShop simply to create the sort of 'drop shadow' effect that you can see in Figure 2. In order to make a convincing drop shadow like the one shown here, it's necessary to 'blur' the image to some extent. In real life, when viewing an object in ambient light (as opposed to strong sunlight) the shadow will generally have a soft focus effect, and this is most easily achieved by

➤ *Figure 2: Drop shadows can be drop dead gorgeous! But in order to implement them, you need to be able to programmatically blur an image. Enter the deeply wonderful Gaussian smoothing algorithm...*



using an image processing function known as a Gaussian blur.

You're probably familiar with the Gaussian distribution so beloved of statisticians, and you might wonder exactly what Gaussian probability curves have got to do with blurring a bitmap. The answer is a lot. If you imagine an aerosol can spraying paint through a tiny pinhole onto a sheet of paper, it should be obvious that you'll get a high density of paint directly opposite the pin-hole, with a gradual reduction in paint density as you get further away. This is a Gaussian distribution. Now consider how an image blurring function should work. Let's pick a pixel at random and consider what colour that pixel should have once the image has been blurred. Obviously, it will be most influenced by those pixels closest to it, and least influenced by those pixels far away. In order to implement this, we need to apply a 'weighting' to each pixel, so that the 'influence' of other pixels falls off smoothly, the further we get from our target pixel. This is where the Gaussian distribution is most appropriate. By creating a weighting array which approximates a Gaussian distribution, we can easily implement a convincing 'blur' function in Delphi.

A particularly nice property of the Gaussian blur algorithm is the fact that it can be applied in a one-dimensional manner. What this means in real terms is that we can blur all the rows in a bitmap, and then blur all the columns. Why is this important? It's crucial because if the convolution (a technical term for the image processing function) had to be applied in a two-dimensional manner, then the processing time would be proportional to X multiplied by Y where X and Y are the width and height of the bitmap. However, since it can be applied in a one-dimensional manner, the processing time will be proportional to X + Y which is obviously going to give a drastic improvement in performance. Against this, there's the overhead of setting up the weighting array, but this typically is an insignificant overhead compared to the convolution itself.

The code for applying a Gaussian blur to a bitmap is included in Listing 1. In fact, I used this code to create the effect shown in Figure 2. I have to confess that the blurring code isn't entirely my own work: I found a public domain implementation on the internet and then extensively reworked it for the purposes of this article. If you want to see the original code that my blur function is based on, then visit:

```
www.delphidevelopers.com/
  delphi/faq/uddf/pages/
  graphics.htm#graphics9
```

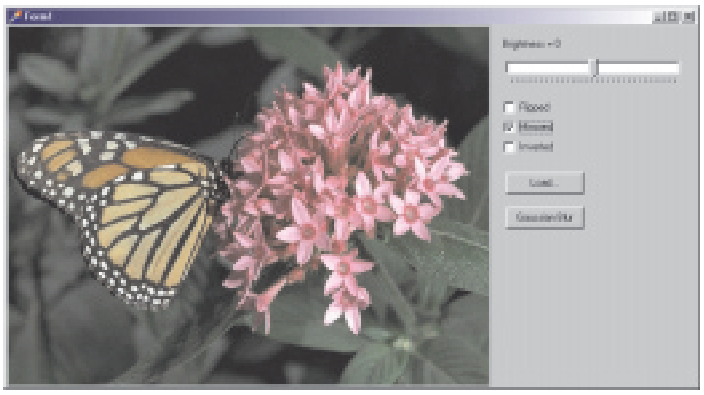
I've considerably altered my version of the code and, after some judicious optimisation, I managed to get a considerable performance increase. The original implementation required 920 milliseconds to blur a particular test bitmap, but my reworked implementation gets this down to 400 milliseconds *[That's quite enough bragging for now, clever clogs. Ed]*.

I should warn you that, as with all Gaussian blur implementations, runtime will rise dramatically as you increase the blur radius. Effectively, you're increasing the width of the 'aerosol spray' and forcing the algorithm to take far more neighbouring pixels into account when calculating the colour of any one pixel. In this implementation, the weightings are stored in an array that's two hundred entries in size, imposing an absolute limit of 100 on the blur radius. However, you will die of boredom long before you get anywhere near this limit. Moreover, it should be clear that convoluting an image with a very large blur radius will simply create an indistinguishable 'smudge', so there's not much point in doing it anyway. Think of blur radius as being synonymous with short-sightedness and you'll get the idea! The code here uses a default blur radius of 3.0.

The heart of the implementation is the `InitGaussianWeightings` routine which builds a Gaussian distribution, with standard deviation equal to the currently specified `blurRadius`. As you'll see, the code simply does nothing if we're using the same blur radius as we did last time, but this optimisation saves us very little time. The weightings array is then normalised so that all the weightings add up to one (the paint has got to land on the paper, at some point!) and the lower and upper bounds are clipped so as to exclude entries that aren't significant before normalising again. Once this is done the main routine, `GaussianBlur`, retrieves the addresses of all the scanlines in the bitmap and saves them in the `Rows` array. Again, this optimisation saves a little time, but not a great deal. Don't be tempted to retrieve the address of the first scanline and then calculate the address of the next by adding the size of a scanline. If you do this, you'll quickly discover that some bitmaps are stored with the first scanline at the highest memory address, with subsequent scanlines preceding it.

The code then walks through each row of the bitmap calling the `BlurRow` routine and copying the newly 'blurred' scanline back into the bitmap proper. It then resizes the `Scratch` buffer so as to hold a column of pixels rather than a row, assembles each column in turn and then calls the `BlurRow` routine to effectively blur each column, copying it back into the bitmap as before.

If you want to speed up this code, your first priority should be the `BlurRow` routine which is where this Gaussian smoothing algorithm spends most of its time. Here, the various weightings are applied to each pixel, taking care to pin the index into the source scanline so that we don't accidentally step outside the bitmap. Again, think of the aerosol analogy here.



➤ *Figure 3: Here's the simple testbed app I wrote to demonstrate the capabilities of this month's code: full source is on the disk as usual.*

Conclusions

Putting all of this together, this month's cover disk includes a simple program that you can use to try out the bitmap manipulation functions covered here. The disk also includes a compiled version of the program (a mere 19Kb) which was written in Delphi 3 using package support. You'll need VCL30.DPL in order to run the EXE file included on the disk. The code has also been tested under Delphi 4. You can see the testbed program running in Figure 3.

While you're playing with this code, you will notice one small problem. If you alter the brightness of an image *after* doing a blur, you'll see the image instantly 'deblur' and snap back into focus. This happens because (as with brightness changes) blurring is a lossy operation, and I don't therefore copy the blurred bitmap into `fOriginal`. When you next change the image brightness, a fresh (non-blurred) copy of the image is retrieved from `fOriginal` cancelling the blur operation. As a quick fix to address this problem, I suggest adding a `Commit` procedure to the `TExBitmap` class, which makes the last change permanent by copying the current bitmap image into `fOriginal`, but depending what you want to do with the code presented here, you may have your own ideas about implementing undo facilities and so forth.

As I said earlier, I hope this article has convinced you that it's quite easy to do clever bitmap processing tricks using Delphi without having to resort to external DLLs or OCX controls. As regular readers will know, I'm a great believer in doing the job in Object Pascal where at all possible. That said, I'd be the first to admit that the image processing tricks shown here are only a start, there are many more effects that would be needed in a real world program. So, over to you! If you've got Delphi source code for any other nice effect such as sharpening, edge detection, de-speckling, motion blur, etc, then I'd love to hear from you.

Dave Jewell is a freelance consultant, programmer and technical journalist specialising in system-level work. He is Technical Editor of *Developers Review* which is also published by iTec. Email Dave at Dave@HexManiac.com