Optimising Display Updating

by Mike Scott

With its optimising compiler and efficient VCL class library, Delphi produces applications that run much faster than those from interpreted products such as Visual Basic. However, the methodology employed in VCL for screen updates has been simplified to ease programming. The downside is that it can be inefficient under certain circumstances.

In this article I will introduce you to two techniques to optimise screen updates which require only a few simple additions to your painting code. To illustrate this I have a written a sample application, FASTDRAW, which is included on the free disk you'll receive with Issue 2. As a bonus, I have also used code that illustrates the use of Windows complex regions for painting, exception handling to protect allocations of Windows resources and how to copy areas directly from and to the screen.

So to begin with, let's go back to first principles. When a window is created, or revealed by moving another window, for example, the parts which were previously obscured and have now been made visible need to be painted. We call these areas invalid regions.

Windows maintains a list of invalid regions and responds to the need for updates by sending messages to the appropriate windows to tell them to repaint themselves. In Delphi applications, the VCL receives this message for you and calls the `Paint` method of your component or form. If you have installed an `OnPaint` handler this is called too.

However, Windows supplies extra information to help you optimise the update but VCL does not pass this on to the `Paint` method or handler. It is non-essential information, your windows will look fine without it. The problem is that they may repaint much slower than necessary because in many cases only part of the window needs repainting. But because VCL doesn't tell you which part, you just have to code your paint method to paint everything. What you need is that extra information.

Fortunately, the Windows API is fairly helpful in this respect. There is a simple function which you can call to get a `TRect` that completely surrounds the invalid region. Then, all you need to do is check which parts of your form or component intersect that area and paint those. It may sound complicated but it's not. It's time to look at the sample.

Ellipses, Ellipses, Everywhere

The sample program simply creates a random pattern of ellipses of different colours spread out to cover an 800 by 600 pixel form. However, when the form first appears it is considerably smaller than this. The form has a toolbar with a checkbox that switches optimised painting on and off. When it's off, the paint method blindly paints every ellipse whether it needs to or not. When the form is in its initial size, for example, only about a quarter of the total number ellipses are visible, but it tries to draw all of them anyway. Of course, Windows makes sure that the ellipses outside the window don't appear, but it still does all the calculations, which wastes a lot of time. For a comparison see Figures 1 and 2.

I have defined a `TEllipse` class, shown in Listing 1. Notice the `Rect` field. It's a good idea to add a rect to classes that you are going to display so that you can quickly determine if they need to be drawn. `TEllipse`'s `Paint` method simply sets the line and fill colours and calls `TCanvas.Ellipse`.

Now let's look at the `Paint` -handlers for the form. I 've written two of these, one "dumb" and the other "smart". When you change the optimised checkbox, the appropriate handler is assigned to the form's `OnPaint` property. The "dumb" paint method simply iterates through the list of ellipses and calls their `Paint` methods as declared above. The loop looks like this:

```
for i := 0 to
  Ellipses.Count - 1 do
    TEllipse(
    Ellipses[i]).Paint(Canvas);
```

When you check optimised, the other, "smart", `OnPaint` handler is assigned. This has additional code to optimise painting. It does this by calling the Windows procedure `GetClipBox` which gets the `TRect` that encloses the invalid region. Then it iterates through the ellipses as before but uses the `IntersectRect` function to check if any part of each ellipse intersects the invalid rect to see if it needs to call `TEllipse.Paint`:

```
GetClipBox(Canvas.Handle,
  ClipRect);
for i := 0 to
  Ellipses.Count - 1 do
  with TEllipse(Ellipses[i]) do
    if IntersectRect(ARect,
    Rect, ClipRect) <> 0 then
      Paint(Canvas);
```

When you get your disk, try running the sample. Press the 'Repaint all' button with and without optimised drawing and note the difference in the times displayed on the toolbar. On my system I get a twelve-fold increase in speed! Interestingly, the time to

Listing 1

```
type
  TEllipse = class( TObject )
  protected
    LineColor : TColor;
    FillColor : TColor;
  public
    Rect       : TRect;
    constructor Create(
      const ARect : TRect;
      ALineColor  : TColor;
      AFillColor  : TColor);
    procedure Paint(ACanvas :
      TCanvas); virtual;
  end;
```

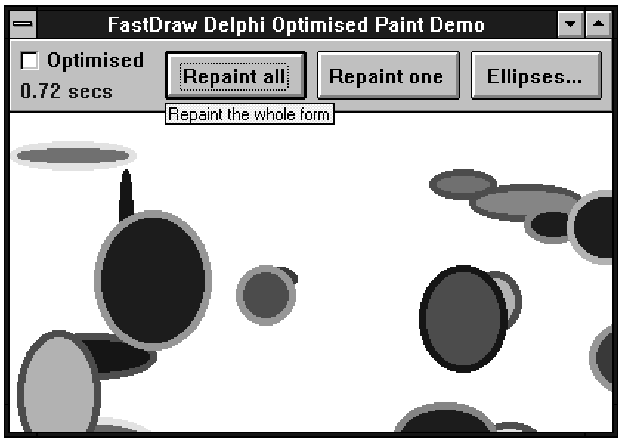

Figure 1

Repainting all the ellipses **without** optimisation

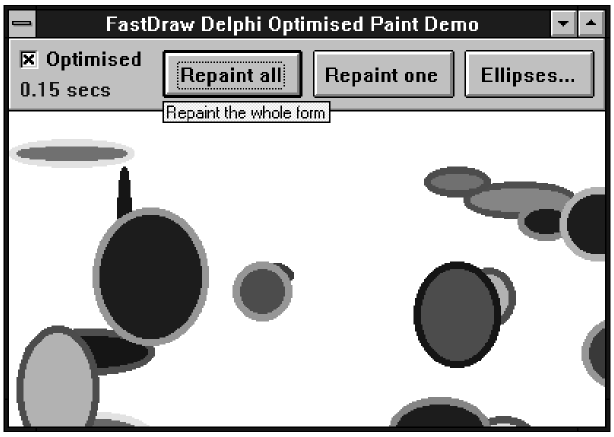

Figure 2

Now notice the difference in repaint time **with** optimisation!

The comparison when only one ellipse is repainted (Repaint one) is even more staggering.

repaint the form using the dumb technique is substantially longer when it is not maximised because of the extra calculations which Windows performs to clip each ellipse to the visible region. In contrast, the optimised method takes less time as the form is reduced in size.

If you maximise the running form, there is virtually no speed difference between dumb and smart redraws because all the ellipses have to be drawn in both cases anyway. The good news is that the additional code that is executed in the optimised case is so fast that you shouldn't notice any increase in the time recorded.

If you press the 'Ellipses...' button you can change the number of ellipses on the form. You will really notice the difference with a large number, say 500 or more. Also, you should try moving the 'Ellipses...' dialog around and noting how long it takes to repaint the revealed area.

You may notice that there is no repaint if you simply close the dialog without moving it. In this case, Windows has decided to take a copy of the background and replace it when the dialog is closed. Moving it forces a repaint and Windows discards the noted area.

Reducing Flicker

VCL has another simplification that can cause another type of annoyance – flicker.

Windows provides a function to invalidate an area of a window to force a paint message to be sent. VCL, however, supplies an invalidate method but this invalidates all of the component or form and tells Windows to erase the background before painting. This erase and then paint causes excessive flicker. A much better way is to erase only the rect that you want to redraw.

The ellipse sample program has an example of this. When you press the 'Invalidate one' button, a single ellipse is chosen at random and invalidated. This causes Windows to send a paint message and the optimised handler only updates the rect for that ellipse. So if you only need to update a part of your component or form, instead of calling its `Invalidate` method, call the Windows API `InvalidateRect` function instead. Here is the sample code that does this:

```
var InvalidRect : TRect;
begin
  ...
  InvalidRect :=
    TEllipse(
      TempList[Random(
      TempList.Count )]).Rect;
  InvalidateRect(Handle,
    @InvalidRect, true);
```

It's not necessary to copy `Rect` into `InvalidRect` but I did so for clarity. `InvalidateRect` takes three parameters: the first is a window handle which you can get from the form or component's `Handle` property. The second is a pointer to a `TRect`, so remember to prefix it the '@' or you'll get 'Error 26: type mismatch' when you try to compile. The last parameter is a boolean that tells Windows whether to erase the background or not. You should generally set this to true. Setting it to false can produce some unusual effects and is not recommended until you know what you're doing!

Regions

As I said at the start, I'll give you a bonus by including some region handling code. You might notice that I draw a thick frame around the ellipse when you click the 'Invalidate one' button to attract your attention to the area being drawn. The code inverts the frame and then inverts it again when the drawing is finished which restores the screen to its original state, preventing the need for invalidation and repainting.

I achieve this by using a region. This is an area which can be any shape and can include holes and gaps. There are a number of Windows API functions which you use to create a complex region by different combinations of simpler regions. To create the frame effect, first I create a `TRect` that is the size of the outside edge of the frame and use `CreateRectRgn` to create a region from this. I do the same for the inner edge and I have two rectangular regions, one defining the outer edge of the frame and the other defining the 'hole' in the middle. I then use the CombineRgn function with the RGN_DIFF operator which gives me a region which is the difference of the two. This effectively removes the 'hole.' I can then invert the region using the InvertRgn function. The code is in Listing 2.

Note the use of try blocks to protect the allocation of the region resources which Windows will not free automatically, even after the program quits. If you're not careful when writing Windows code you can end up with severe resource leakage. A good habit to get into is to automatically type a try statement on the line following any allocation or operation which you have to undo or tidy up later. I then often type in the finally...end with the cleanup code before I even put in the rest of the lines. That way I am sure it won't be forgotten and it's easier to follow the indentation!

I use try...finally to deallocate the two source regions because they must always be freed whether there is an exception or not. I use the try...except block to free up Result only when there is an exception. In most cases you should call Raise at the end of your try... except block to pass the exception back up the stack. On the other hand, you don't call Raise in a finally block because the appropriate processing continues anyway. Region functions generally make a copy of any region passed as a parameter so remember to free up the source regions as I have done in the example.

One very powerful use of regions is to control the clipping area when painting. Windows allows you to specify your own region where painting will be allowed. In the above example, I could have selected the region as the clipping region and then inverted the whole form with:

```
 InvertRect(Canvas.Handle,
    Rect(0, 0, Width, Height))
```

instead of using InvertRgn. The result would have been the same because Windows would limit the invert, or any other drawing operation, to the area defined by the frame region. This is a very powerful technique which you can use to fill or paint complex shapes.

Direct From The Screen And Back...

You may have noticed that I overlay a rectangular red box containing some text along with the inverted frame when the 'Invert one' button is pressed. When the frame is removed, so also is the text box but there is no time-consuming paint. I achieve this by creating a temporary memory bitmap the size of the box and using its canvas to copy the area straight off the screen. To remove the box, all I need to do is copy the area back when I'm finished.

Performing this update trick is actually very simple. You use the canvas CopyRect method which does all the work. All you need to do is create a bitmap, set its width and height and use this method to grab the pixels from the screen. When you're done you use CopyRect in the reverse direction to put the pixels back again and then just free the bitmap. Simple! The code is in Listing 3.

DestRect is a TRect that defines the area on the form on the screen. In this case the other canvas used in CopyRect is that of the form, but it could be any other canvas. Again, I use try...finally to make sure ABitmap gets freed at the end.

You can use a similar technique to completely banish flicker altogether. Instead of painting straight on to the form's canvas, you create a bitmap just like the above code fragment, set the width and height to the width and height of the area you're updating on the form, paint to the bitmap's canvas and then use CopyRect to blast the result straight on to the screen with no trace of flicker at all. Because you are not going across a bus to the screen card with every drawing operation, this technique is often faster than the usual method of writing direct to the form's canvas. Space does not allow a proper example or full details. That is the topic for another article...!

Listing 2

```
function
CreateFrameRegion(const ARect :
  TRect) : HRgn;
var Region1, Region2 : HRgn;
begin
  { creates a "frame" area using
    regions as an illustration -
    also illustrates protecting
    code with try blocks }
  with ARect do begin
    Region1 :=
      CreateRectRgn(Left - 6,
        Top - 6, Right + 6,
        Bottom + 6);
    try
      Region2 :=
        CreateRectRgn(Left, Top,
           Right, Bottom);
      try
        Result := CreateRectRgn(
          0, 0, 0, 0);
        try
          { remove region 2 from
            region 1 and delete
            the source regions }
          CombineRgn(Result,
            Region1, Region2,
            RGN_DIFF);
        except
          DeleteObject(Result);
          Raise;
        end;
      finally
        DeleteObject(Region2);
      end;
    finally
      DeleteObject(Region1);
    end;
  end;
end;
```

Listing 3

```
var ABitmap : TBitmap;
begin
  ...
  ABitmap := TBitmap.Create;
  try
    ABitmap.Width :=
      DestRect.Right;
    ABitmap.Height :=
      DestRect.Bottom;
    { grab the pixels from
      the form's canvas }
    ABitmap.Canvas.CopyRect(
      DestRect, Canvas,
      SourceRect);
    { ... do whatever you need
      to do ... }
    { & copy pixels back again}
    Canvas.CopyRect(SourceRect,
      ABitmap.Canvas, DestRect);
  finally
    ABitmap.Free;
  end;
```

Mike Scott is a Director of Mobius Software which specialises in Delphi VCL component tool kits and applications and is based in Edinburgh, Scotland. He can be contacted via CompuServe at 100140,2420 (on the internet it's 100140.2420@compuserve.com), or telephone +44 (0)131-467 3267