# *Beating The System:* A Smorgasbord Of GDI Wonders

*by Dave Jewell*

As all Delphi programmers know, the VCL framework gives us a convenient wrapper around much of Windows, providing a programming model which is far simpler and more convenient to work with than the underlying API. The way in which Borland are currently re-implementing a large subset of the VCL library on top of Qt (the CLX framework which will form part of Delphi for Linux) is a testimonial to the way in which relatively few Windows-specific artefacts were exposed at the VCL programming level. MFC, of course, is a very different kettle of fish.

Having said all that, it can't be denied that there will always be times when it becomes necessary to hit the underlying API in order to accomplish things that couldn't be done with the VCL library alone. In this article, I'm going to look at some of the more interesting parts of the GDI which aren't accessible at the VCL level. Please bear in mind that all the code snippets discussed here have been tested under Windows 2000 only. If you want to try things out on other platforms, your mileage may vary.

## Display Mode Enumeration

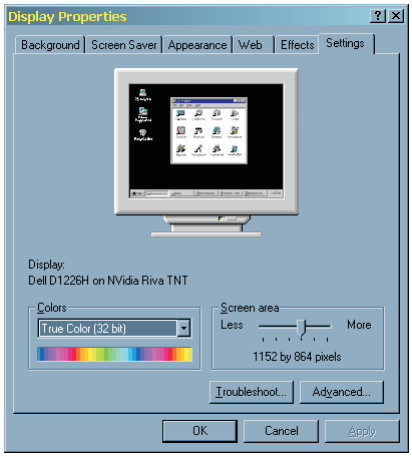
It's sometimes useful to be able to determine programmatically what display resolutions are available, and to switch to a new display resolution under program control. A sophisticated drawing program might offer to switch to a higher resolution if it determines that the current screen layout is going to be cramped, or a screensaver which uses palette animation might need to switch down to a display mode that makes use of palettes. Then again, you might want to write your own version of Microsoft's QuickRes application which sits in the taskbar tray area and allows instant reconfiguration of display mode.

The key to display mode enumeration is a GDI routine called `EnumDisplaySettings`, the function prototype of which is given below:

```
function EnumDisplaySettings
  (lpszDeviceName: PChar;
  iModeNum: DWORD;
  var lpDevMode: TDeviceMode):
  BOOL; stdcall;
```

➤ *Listing 1*

```
procedure TForm1.GetDisplayModes;
var
  Idx: Integer;
  DevMode: TDeviceMode;
  PMode: PDeviceMode;
  Item: TListItem;
begin
  Idx := 0;
  while EnumDisplaySettings (Nil, Idx, DevMode) do begin
    New (PMode);
    PMode^ := DevMode;
    Item := Modes.Items.Add;
    Item.Data := PMode;
    Item.Caption :=
      Format('%d * %d', [DevMode.dmPelsWidth, DevMode.dmPelsHeight]);
    Item.SubItems.Add (IntToStr (DevMode.dmBitsPerPel));
    if not (DevMode.dmDisplayFrequency in [0, 1]) then
      Item.SubItems.Add (IntToStr (DevMode.dmDisplayFrequency) + ' Hz')
    else
      Item.SubItems.Add ('- device default -');
    if SameMode (DevMode, CurMode) then Modes.Selected := Item;
    Inc (Idx);
  end;
end;
```




➤ *Figure 1: It turns out that much of the functionality of the Control Panel's Display Settings applet is relatively easy to emulate, once you know how to enumerate the available display modes on your graphics card.*

The first parameter, `lpszDeviceName`, is used to specify the name of the display device that we are interested in. You can pass a value of `Nil` here in order to indicate the current display device. You can think of the next parameter, `iModeNum`, as an index into the available display modes: pass 0 to get information pertaining to the first display mode, 1 for the next, and so forth. There are also a couple of special values which can be passed via this parameter; these provide information relating to the current display mode and the mode information as stored in the registry. You can find more information on this in the Microsoft SDK documentation.

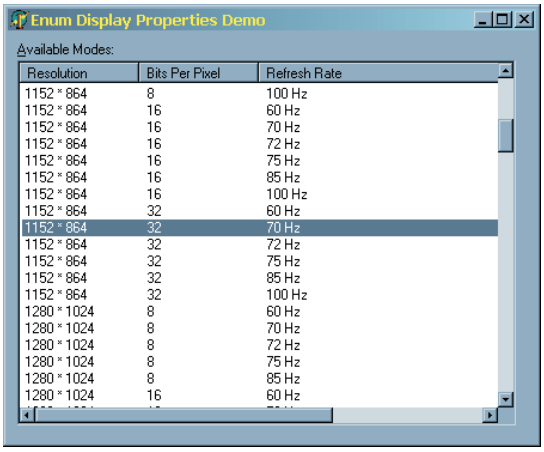
The final parameter, `lpDevMode`, is a reference to a record of type `TDeviceMode` through which all the display information is passed. If you look at the definition for `TDeviceMode` in the WINDOWS.PAS unit, you will probably be surprised at just how much information gets returned here. It also includes entries such as `dmPaperLength` and `dmPaperWidth` which demonstrate that this low-level data structure is used not only to communicate with the graphics adaptor but with printer devices as well. Enumeration continues until a value of 0 is returned from

the call to EnumDisplaySettings, at which point we know that all available display modes have been enumerated. You can see an example of how to use this function in Listing 1.

This code has been taken from a working program, called ENUMDISPLAY.EXE, the complete source code for which is included on this month's cover disk. As you can see, the code repeatedly calls the EnumDisplaySettings routine, passing an index value which increments by one each time around the loop. If a new display mode is found, then a complete TDeviceMode structure is allocated on the heap, and a pointer to this structure is saved into the Data field of the associated TListItem. Of course, this may be over the top for some applications, it's highly unlikely that you'd be interested in more than a fraction of the fields in the TDeviceMode data structure.

The dmPelsWidth and dmPelsHeight fields contain the pixel width and height (respectively) of the display mode. Incidentally, if you're wondering what a 'pel' is, I believe this is a historical hangover from IBM terminology and stands for a 'Picture Element', it's basically just a pixel. The dmBitsPerPel field indicates how many bits are required for storing a single pixel. This will be 8 for 256 colour systems, 16 for High Color (65,536 colours) systems and 32 for True Color (4,294,967,296 colours) displays.

Finally, dmDisplayFrequency gives the vertical refresh rate for the given display mode. In general, the higher the refresh rate, the more flicker-free will be the screen display, but bear in mind that if you push some older monitors too far, you could end up with a very dead monitor! *You have been warned: don't blame me if your monitor dies!* Also, it's worth noting that the dmDisplayFrequency field will sometimes return a value of 0 or 1. This obviously doesn't correspond to a refresh rate in Hertz! Rather, it simply indicates that this display mode uses the default hardware refresh rate of the device. This is reflected in Listing 1.




➤ *Figure 2: Here's the display mode enumerator program in action. Notice that each possible pixel width and height combination has several variations, corresponding to different vertical refresh rates.*

Obviously, it's also nice if your program can determine which of the available display modes is the one currently in use. I did this by writing another routine, SameMode, which tests two display modes for equivalence. It does this by comparing the pixel width and height, colour depth and vertical refresh frequencies. This is shown in Listing 2.

Of course, this raises the question of how the program gets its hands on the current video mode metrics? As a Delphi developer, you could always use Screen.Width and Screen.Height to get the screen dimensions, and a little jiggery-pokery with Canvas.Handle and GetDeviceCaps would get you the number of bits per pixel, but what about refresh rate?

It turns out that is also available from the GetDeviceCaps routine (use an index of VREFRESH) but a simpler alternative is just to pass the special index value of Enum_Current_Settings to the EnumDisplaySettings routine. This will give you a complete TDeviceMode record that corresponds to the current display mode. Unfortunately, I couldn't find the value of Enum_Current_Settings anywhere in Borland's WINDOWS.PAS file (Delphi 5 version) but, after a little hacking around inside Windows, I established that this constant is equal to $FFFFFFFE. This gives us the program initialisation code shown in Listing 3.

You can see the result running in Figure 2. As you'll notice, the program provides an exhaustive list of all the available display modes, together with the colour depth and refresh rate in each case. Notice also that the code has highlighted the current video mode.

## Setting A New Display Mode

So what about the situation where we want to change the display mode? This is done via another little-used GDI routine called ChangeDisplaySettings:

```
function ChangeDisplaySettings
  (var lpDevMode: TDeviceMode;
  dwFlags: DWORD): Longint;
  stdcall;
```

Again, this routine takes a reference to a TDeviceMode data structure which is used to specify the wanted mode. At this point you'll appreciate why I scrupulously cached all those TDeviceMode data structures back in Listing 1. If this

```
function TForm1.SameMode (Mode1, Mode2: TDeviceMode): Boolean;
begin
  Result := (Mode1.dmPelsWidth = Mode2.dmPelsWidth) and
    (Mode1.dmPelsHeight = Mode2.dmPelsHeight) and
    (Mode1.dmBitsPerPel = Mode2.dmBitsPerPel) and
    (Mode1.dmDisplayFrequency = Mode2.dmDisplayFrequency);
end;
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
procedure TForm1.FormCreate(Sender: TObject);
const
  Enum_Current_Settings = $FFFFFFFE;
begin
  EnumDisplaySettings (0, Enum_Current_Settings, CurMode);
  GetDisplayModes;
end;
```

hadn't been done, then it would have been necessary to build a `TDeviceMode` data structure from scratch for passing to the `ChangeDisplaySettings` routine. Well, maybe: it would probably be OK to get the `TDeviceMode` corresponding to the current display mode, make a copy of it, plug the required resolution, colour depth and refresh info into the new data structure and use that, but we may as well do things properly.

Bear in mind that, when specifying a new display mode, you need to give some attention to the `dmFields` field of the `TDeviceMode` record. This is a set of bit flags that tell the GDI which fields of the record are significant when changing mode, as shown in Listing 4.

Because we're setting all four of these values, the `dmFields` mask is set to be the bitwise `OR` of these four constants.

*Warning: because we're specifying a new refresh rate, this does mean you could end up with a dead monitor, as I mentioned earlier. If you've got any doubts about giving this capability to your end-users, then don't include* `DM_DisplayFrequency` *in the* `dmFields` *flag.*

The second parameter to `ChangeDisplaySettings` is another set of bitwise flags which specify the way in which the display mode should be changed. Most typically, you'd pass `cds_UpdateRegistry` as the flag value since this not only changes the display mode, but it also writes the display mode settings to the registry so that the same mode will be used on the next reboot. These display mode settings are stored on a per-user basis, but if you have adequate access privileges, you can include the `cds_Global` which will update display mode settings for *all* users. The `cds_Test` flag causes Windows to test if the specified graphics mode can be set without actually making any change, whereas `cds_NoReset` saves display mode changes to the registry, again, without actually changing the display mode there and then. There are various other possible flag values, which you can read about in the SDK documentation.

Putting this together, we get the code shown in Listing 5. The routine starts off by confirming that the user actually does want to make the specified display mode change. If so, a new `TDeviceMode` record is initialised and the all-important `dmFields` field is set up as discussed earlier. Notice that if no vertical refresh rate was specified in the device mode record, then we take care not to '`OR`-in' the `DM_DisplayFrequency` flag. If it were not for this code, you might find yourself asking the display hardware to use a refresh rate of 1 Hz!

And that's about it as far as display mode changes are concerned. As mentioned above, the complete executable code and source is included on this month's disk.

| | | |
|---|---|---|
| `DM_PelsWidth` | ➔ | The screen width field is significant |
| `DM_PelsHeight` | ➔ | The screen height field is significant |
| `DM_BitsPerPel` | ➔ | The bits per pixel field is significant |
| `DM_DisplayFrequency` | ➔ | The vertical refresh frequency field is significant |

➤ *Listing 4*

I haven't bothered to wrap the code up into a reusable component, although that would be very easy to do. You'll soon discover (as does anyone who changes their display mode settings!) that Windows Explorer routinely mangles the layout of icons on your desktop each time the screen's pixel resolution changes, but that's not my fault. Rather, saving and restoring your desktop layout is something that we'll look at in a future issue.

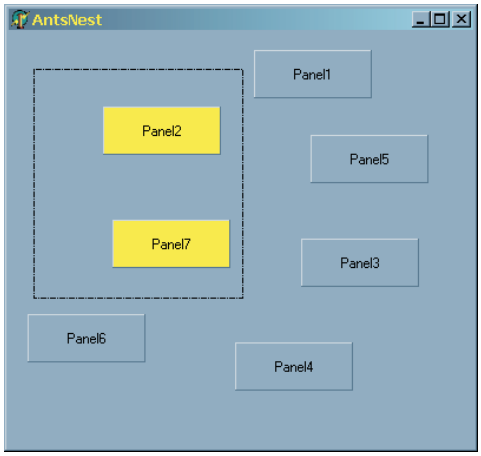
## Crawling Ant Effects

I currently have a large wasps nest in the roof space just above my office, so you can well imagine that I don't much like talking about insects right now! But the 'crawling ants' effect is something that many developers wish to exploit sooner or later, and in this section I will discuss some ways of achieving it.

In case you're not familiar with what I'm talking about, 'crawling ants' (also know as 'marching ants') is a common nickname used to describe the animation effect when a selection rectangle is created in a drawing program. Many programs cheat by simply using a static 'rubber-band' effect, but if you want to do the job properly, then you should consider having an animated selection rectangle

➤ *Listing 5*

```
procedure TForm1.ModesDblClick(Sender: TObject);
var
  Item: TListItem;
  ModeStr: String;
  NewMode: TDeviceMode;
  DevMode: PDeviceMode;
begin
  Item := Modes.Selected;
  if Item <> Nil then begin
    DevMode := Item.Data;
    with DevMode^ do begin
      ModeStr := Format(
        'Switch display mode to %d * %d, (%d bpp',
        [dmPelsWidth, dmPelsHeight, dmBitsPerPel]);
      if not (dmDisplayFrequency in [0, 1]) then
        ModeStr := ModeStr +
          Format(', %d Hz refresh', [dmDisplayFrequency]);
      if MessageDlg (ModeStr + ') ?', mtConfirmation,
        mbOKCancel, 0) = mrOK then begin
        NewMode := DevMode^;
        NewMode.dmFields := dm_PelsWidth or
          dm_PelsHeight or dm_BitsPerPel;
        if not (dmDisplayFrequency in [0, 1]) then
          NewMode.dmFields :=
            NewMode.dmFields or DM_DisplayFrequency;
        case ChangeDisplaySettings (NewMode,
          cds_UpdateRegistry) of
          Disp_Change_Successful:
            begin
              ShowMessage('The display mode has been '+
                'successfully changed');
              Modes.Selected := Item;
            end;
          Disp_Change_Restart:
            if MessageDlg('Need to reboot for display '+
              'mode change to take effect.  Do it now?',
              mtConfirmation, mbOKCancel, 0) = mrOK then
              ExitWindowsEx (ewx_Reboot, 0);
          Disp_Change_BadMode, Disp_Change_Failed:
            ShowMessage ('Trouble at ''mill: display mode '+
              'change failed or not supported');
        end;
      end;
    end;
  end;
end;
```




➤ *Figure 3: You might not be able to see it, but the selection rectangle in this screenshot is animating as the 'ants' crawl around the borders of the selected area. The highlighted panels are simply used to indicate what is selected and what is not.*

which looks as if a line of ants are slowly marching around the borders of the rectangle.

The key to the 'crawling ants' effect is a somewhat weird GDI routine called `LineDDA`, the function prototype for which is:

```
function LineDDA(
  X1, Y1, X2, Y2: Integer;
  Proc: TFNLineDDAProc;
  Data: Integer): BOOL;
  stdcall;
```

When 'Pascalising' Windows API routines, Borland generally give sensible names to the function arguments. However, `LineDDA` is one that never received this honour. Accordingly, the above function prototype has been slightly massaged by me for the sake of clarity, compared with the original in WINDOWS.H.

The first four arguments specify the start and end points of a line that's to be drawn by the function. You'll notice that in this case, no device context is supplied to `LineDDA`, which is obviously very unusual for a line-drawing routine! The reason is that we (ie, the application program) are responsible for drawing the lines rather than Windows itself. For each calculated point on the line, the GDI calls an application-supplied routine to perform the actual drawing. By getting in on the act on a per-pixel basis, this gives us the possibility of creating custom line effects and line animations.

The fifth parameter to `LineDDA` is a pointer to the call-back routine, the function prototype for which is given below:

```
procedure LineDDAProc(
  X, Y: Integer;
  Data: Integer); stdcall;
```

As with other callback routines, this must be declared as a `stdcall` routine in order for it to be properly called from Windows. The `X` and `Y` arguments obviously correspond to a specific point on the line, whereas the `Data` argument corresponds to the final argument to `LineDDA`: it enables us to pass an application-defined value which is accessible to the callback routine. Since Windows API callback routines can't be methods of Delphi objects, this is a great way of recovering the 'OOP context' within our callback.

The sample program that I developed for this article, CRAWLINGANTS.EXE, was inspired by an article in the September 1996 issue of *The Unofficial Newsletter Of Delphi Users*. My program works in basically the same way, but it's a considerably simplified and optimised version of the original code, which you can find at www.undu.com/DN960901/00000008.htm.

You can see the program running in Figure 3. Unlike the original, I've used panel components as the 'selectable objects' in the form window. Whenever you use the mouse to select one or more panels, the selected items turn yellow to provide a visual indication of their selected state. At the same time, the selection rectangle begins animating to show the selected region. In a real world drawing program, each of the items on a form would typically be a Delphi object, and you might provide a virtual method by means of which the selection state of the

➤ *Figure 4: This is a close-up of the dot/dash that results from using a mask value of $A0. See the text for more details of how to come up with other dot/dash patterns.*

➤ *Above: Listing 6*

```
procedure TAntsNest.DrawHotRect;
const
  StartMask: Byte = $80;
begin
  StartMask := StartMask shr 1;
  if StartMask = 0 then
    StartMask := $80;
  DashMask := StartMask;
  with HotRect do begin
    LineDDA (Left, Top, Right, Top, @LineDDAProc, Integer (Self));
    LineDDA (Right, Top, Right, Bottom, @LineDDAProc, Integer (Self));
    LineDDA (Right, Bottom, Left, Bottom, @LineDDAProc, Integer (Self));
    LineDDA (Left, Bottom, Left, Top, @LineDDAProc, Integer (Self));
  end;
end;
```

➤ *Below: Listing 7*

```
procedure LineDDAProc (X, Y: Integer; Self: TAntsNest); stdcall;
const
  DotPattern: Byte = $a0;
var
  C: Integer;
begin
  with Self do begin
    DashMask := DashMask shl 1;
    if DashMask = 0 then
      DashMask := 1;
    if (DashMask and DotPattern) <> 0 then
      C := Color
    else
      C := clBlack;
    SetPixel (Canvas.Handle, X, Y, ColorToRGB (C));
  end;
end;
```

object could be turned on or off. It would then be up to the individual object (be it a rectangle, ellipse, integrated circuit, dining table, or whatever) to show its selection state in an appropriate manner. The Delphi IDE simply draws 'grab handles' around selected items on the design-time form.

The speed at which the selection rectangle animates is controlled by the `TTimer` component which drives the animation. I chose an interval of 50 milliseconds or 20 times per second. The timer's `OnTimer` event handler simply calls the routine shown in Listing 6.

Each time this routine gets called, we want to redraw the selection rectangle, but with the 'ants' having advanced by one footstep, so to speak! This is controlled by the `StartMask` variable which starts at `$80`, and progressively shifts down to 1 at which point it's reinitialised to `$80`. This effectively gives us eight possible 'states' that the selection rectangle can be in. For each of those states, we need to start drawing the differently coloured dashes at a progressively different pixel position so as to get the effect of smoothly marching ants.

The current selection rectangle is stored in the `HotRect` variable, and this is used to issue four separate calls to the `LineDDA` routine, one for each side of the rectangle. As you'll see, `Self` is passed as the final parameter to `LineDDA` in each case, as well as specifying `LineDDAProc` as the actual drawing routine.

Inside `LineDDAProc` (Listing 7), the `DashMask` variable is used within the drawing of each line (remember, `DrawHotRect` gets called on each timer tick to draw the entire rectangle, but `LineDDAProc` is called for each pixel that needs to be drawn) to decide whether each pixel should be black or the same colour as the form. The original code effectively used a value of `$E0` for the `DotPattern` mask, but if you experiment with different values of this mask, you'll be able to get more interesting patterns of dots and dashes. A value of `$A0`, for example, gives you a nice big ant followed by a baby one! You can

## Digging Deeply Into Delphi DCUs: An Update

My, how time flies! It's now been almost 18 months since my original three-part article in *The Delphi Magazine* which discussed the internals of DCU files. Although the information I presented was admittedly incomplete, I can say that, as far as I know, nothing else had previously been published which attempted to describe the DCU file format in any detail. Since those articles were published, a number of other folks have taken the understanding of the DCU file structure a good deal further, and I flatter myself that I had some small part to play in making this happen. If you want to catch up with the free tools that are available, here's a snapshot of the current state of play.

Probably the most impressive Delphi Decompiler around at the moment is DeDe, which of course stands for 'Delphi Decompiler'. You can find DeDe at www.balbaro.com. Figure 5 shows an example of DeDe peeking inside a Delphi executable, in this case the Image Editor program which comes as part of the Delphi package. As you'll see from the screenshot, DeDe has identified all the classes (not just form classes) used inside the program, along with all the units that are linked into the code. DeDe provides another view which lets you see the various .DFM files as human-readable text, which can then be pasted into an IDE design-time form while in 'View as Text' mode. This essentially makes it possible to extract a complete Delphi form from a finished application, although you'd obviously need to recreate any associated event handlers.

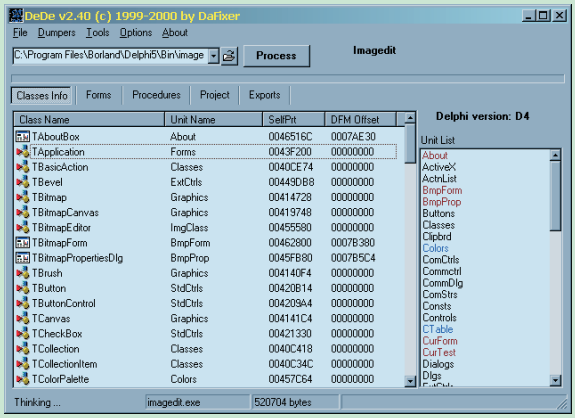



*Figure 5: DeDe is a Delphi decompiler which can provide a frightening amount of information about the internals of your apps. Parts of DeDe are based around the DCU32INT utility which understands the internal format of Delphi's 32-bit DCU files.*

Perhaps the most impressive feature of DeDe is the ability to list all the event handlers associated with a certain form, whereupon double-clicking an event handler shows you an assembly language dump of the relevant code. DeDe includes code signature files for the various VCL libraries (VCL40, VCL50, etc) making it possible for the utility to automatically recognise specific library calls in the disassembled executable. This is akin to the so-called FLIRT code recognition capabilities of IDA Pro (www.datarescue.com). One thing which DeDe has but IDA Pro lacks is the ability to automatically recognise `try..finally` and `try..except` blocks in the disassembled code. DeDe can also regenerate the .DFM, .PAS and .DPR files associated with a project, but obviously, the .PAS files only contain assembler code! The program will also generate symbolic reference files which can be used with IDA Pro, or with another popular disassembler, W32DASM.

But what of DCU files? DeDe includes a couple of utilities for dumping .BPL/.DPL files and .DCU files. A disassembled DCU file looks perhaps a little messier than it might; for example, the implicit 'this' parameter to methods of a class is listed as a formal parameter when dumping method bodies, but nevertheless DeDe is a great time-saver for those who enjoy poking around 'under the hood'. The software is freely downloadable, although it's a shame the author hasn't released source code.

Truth to tell, it matters not a jot that the DeDe author hasn't released source code. Why? Because the DCU-munching part of DeDe is based around DCU32INT, and complete source code for *that* is available. If you need more information on the innards of DCU files and want to write DCU disassemblers for yourself, take a look at http://monster.icc.ru:8080/~alex/DCU/index.eng.html which is the home of DCU32INT. It's so named because it takes a 32-bit Delphi DCU file and spits out an .INT file which describes the interface part of the unit and generates assembler code corresponding to the implementation part. (Apparently DCU32INT came out of the FlexT project which formed the author's doctoral thesis. FlexT is a way of describing the internals of an arbitrary binary file in a formal way.) DCU32INT is fast, free, recovers a lot of interesting information from a DCU file and the price is definitely right! If you know of any other DCU peeking utilities, please let me know.

see this clearly in Figure 4 which shows a magnified view of the resulting selection rectangle.

Other effects can be easily achieved. For example, if you want to have a double-thickness selection rectangle, then you could add the following statement to the `LineDDAProc` routine immediately after the existing `SetPixel` call:

```
SetPixel(Canvas.Handle, X+1,
  Y+1, ColorToRGB(C));
```

This would draw an additional pixel immediately below and to the right of the existing one. With this approach, you'd also have to modify the code inside `SetHotRect` which is responsible for erasing the existing selection rectangle whenever a new one is drawn. In fact, if you wanted to go the whole hog and create a general purpose, reusable, component to implement an animated selection rectangle, then it would be a good idea to implement a `LineThickness` property together with a `DotPattern` property to specify what pattern of dots and dashes are required. Needless to say, this is left as an exercise for the reader!

Another approach, favoured by Mark Miller of CodeRush fame, is to use a completely different way of drawing animated selection rectangles. In CodeRush, Mark uses a series of carefully designed bitmap masks which are combined with the current screen content and then blitted onto the canvas. This is a more complex approach, but arguably gives more professional results, provided you don't mind having a fixed line thickness in your selection rectangle.

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