Beating The System: Alpha Blending In Windows 2000

Clearly Minty!

by Dave Jewell

I don't know how long you've been working with Windows, but I've been at it ever since version 1.0. There's one particular programming experience which will always stick in my mind, and this goes back to the days when EGA cards (Enhanced Graphics Adapter) ruled the roost. In case you're not as long in the tooth as me, suffice it to say that an EGA card gave you a princely resolution of 640 by 350 pixels in no less than sixteen glorious colours. And if you think *that's* bad, you really wouldn't want to hear about the predecessor to the EGA card!

But I digress. My memorable programming experience happened over 15 years ago when I was debugging some long-forgotten Windows application using a very crude (by today's standards) kernel mode debugger. The program being debugged appeared on the main monitor, whereas the text-mode debugger had a monochrome monitor all to itself, where you entered cryptic one-character commands followed by a string of hexadecimal gibberish. I can't remember exactly what I did wrong, but somehow I must have entered something that contained even more gibberish than the debugger was expecting. When I hit the `G` key to run the program, something wonderful happened.

At first, everything looked normal, until the moment when I clicked the mouse on a menu bar item. Instead of a drop-down menu appearing in the normal fashion, I suddenly saw a beautiful translucent menu appear, with the previous screen contents tastefully dimmed behind it. My jaw crashed to the floor as I moved the mouse around my magical menu, transfixed in wonder (come on, give me a break, this was 1985!). I don't know how or why it happened, but somehow I managed to get that ancient EGA card into a mode where it was layering the popup menu on top of the existing screen content. It lasted for all of thirty seconds before Windows disappeared up its own rear end and I found myself giving the keyboard the usual three-fingered salute. I was never able to reproduce the incident, and it left me a broken man. I had seen paradise, and could never regain it.

Paradise Regained

Until now, that is! One of the joys of Windows 2000 is the ability to create application windows that are translucent, using alpha blending. If you're a dedicated API watcher, you may have noticed that Microsoft added some basic alpha blending code to Windows 98 through the new `AlphaBlend` call. However, neither Windows 95 nor Windows 98 contains operating system support for the creation of translucent windows.

Before proceeding, let's be very clear about our terminology. By *translucent* I'm talking about a window which is partially opaque, allowing a somewhat dimmed version of the background screen contents to show through, just like a piece of coloured cellophane, or whatever. A *transparent* window, on the other hand, has zero opacity. It has no effect on the visual appearance of whatever is behind, so it's effectively invisible; or, at least, the transparent parts of it are!

Under Windows 95/98, it's possible to create a window which *appears* to be partially translucent or partially transparent, but the runtime overheads in doing this are often very high. For example, you can easily use `SetWindowRgn` to create a non-rectangular window, but as you move such a window around the screen, the operating system has to do a lot of work in calculating appropriate update regions for each of the underlying windows. Similarly, it's possible to create translucent windows under Windows 95/98, but the runtime performance is less than spectacular, because the application has to take responsibility for figuring out what's 'underneath' it, dimming the bitmap, and updating everything in real time as the window moves around the screen.

With the introduction of Windows 2000, all this has changed through the introduction of a new concept called *layering*. This enables the operating system to take responsibility for buffering the contents of the window behind the current window. If you find that hard to understand, let's put it in concrete terms. Suppose your program has a window, A, in front of another application's window, B. Under Windows 95/98, if you move A so as to expose more of B, or close A altogether, Windows has to send a message to B telling it to repaint the newly uncovered area. This is the traditional functionality that most Windows developers are familiar with. However, under Windows 2000, you can make your window, A, into a layered window. This causes the operating system to take a 'snapshot' of the screen behind A. Anytime that the background behind A needs to be repainted, Windows 2000 can simply do the job itself using the contents of its off-screen buffer. This is obviously a lot faster than asking the application to do it and drastically reduces the flicker

```
procedure TForm1.CreateParams (var Params: TCreateParams);
begin
  Inherited CreateParams (Params);
  Params.ExStyle := Params.ExStyle or ws_Ex_Layered;
end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
function SetLayeredWindowAttributes (Wnd: hWnd; crKey: ColorRef; bAlpha: Byte;
  dwFlags: DWord): Bool; stdcall; external 'user32.dll';
```

which typically characterises non-rectangular windows, or attempts to get translucency, under Windows 95/98.

This explanation is somewhat simplistic. If you think about it, Windows also has to take a snapshot of the layered window in its fully opaque state. There are thus two off-screen bitmaps: the image of the window and the image of whatever is behind the window. These are combined together using alpha blending to create the translucency effect.

Layering is applicable not only to translucency, but also to animation effects. Whenever you're doing anything that's going to involve much updating of the window(s) behind your application window, then it makes sense to use layering.

Translucent Application Windows

OK, enough theory, let's roll up our sleeves and have some fun. Before we can make an ordinary application window translucent, we have to tell Windows that it's a layered window. This is done using a new extended window style, WS_EX_LAYERED. From the viewpoint of the VCL and Delphi programming, you could set this style bit immediately before the window is created by overriding the CreateParams routine, as shown in Listing 1.

This will work, but it's unnecessary. There's no need to override CreateParams simply to set this bit because, unlike some other style bits, Windows will happily allow you to alter the style bit after the window has been created. A simpler approach is to use the GetWindowLong and SetWindowLong routines to massage the style bit as and when needed. If you try this out with a bare-bones Delphi program, you might be disappointed to see that your form doesn't appear at all! That's because we've told Windows that the form is a layered window, but we haven't specified the required opacity or transparency characteristics.

The missing piece of the jigsaw is a new routine called SetLayeredWindowAttributes. Not surprisingly, the function prototype for this routine hasn't made it into Delphi yet (not even Delphi 5) but Listing 2 is one I prepared earlier.

As you can see, this routine lives in the USER32 library. This call will let you set opacity or transparency information for the layered window specified by the Wnd handle. There are two possible values that can be passed to the dwFlags parameter, as shown below:

```
// LWA constants for
// SetLayeredWindowAttributes
lwa_ColorKey   = 1;
lwa_Alpha      = 2;
```

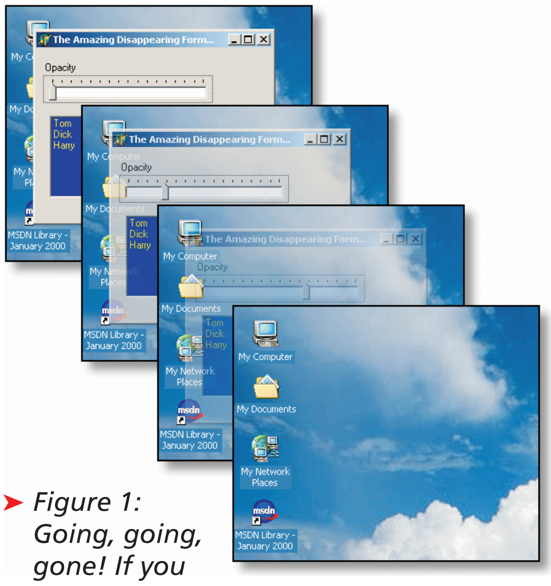
You pass a value of lwa_ColorKey to set the colour key (transparency) attribute, or you can pass lwa_Alpha to set the translucency (opacity) attribute. In the former case, the information is passed in the ColorRef parameter, and in the latter case through bAlpha.

Since I've been waxing lyrical about translucency, let's deal with that first. As you can see, bAlpha is a byte, allowing us to specify an opacity value that varies from 0 to 255. A value of 0 means that the window is totally non-opaque or, to put it another way, is completely transparent! If you use an opacity value of zero, then the window will be totally invisible. On the other hand, a value of 255 means that the window is totally opaque, and looks just like a normal window.

The interesting stuff, of course, lies somewhere between. Listing 3 shows the complete source for a tiny Delphi program which allows you to change its opacity on the fly. This is done through an ordinary slider control which calls SetLayeredWindowAttributes every time that its position is changed. You can see the result of running this program in Figure 1, as the slider moves over to the right, the opacity is decreased until the form

➤ *Listing 3*

```
unit ClearForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ComCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    CheckBox1: TCheckBox;
    ListBox1: TListBox;
    TrackBar1: TTrackBar;
    Label1: TLabel;
    procedure FormShow(Sender: TObject);
    procedure TrackBar1Change(Sender: TObject);
  private
  public
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
const
  // LWA constants for SetLayeredWindowAttributes
  lwa_ColorKey        = 1;
  lwa_Alpha           = 2;
  // New extended window style for layering
  ws_Ex_Layered       = $80000;
function SetLayeredWindowAttributes (Wnd: hWnd; crKey:
  ColorRef; bAlpha: Byte; dwFlags: DWord): Bool; stdcall;
  external 'user32.dll';
procedure TForm1.FormShow(Sender: TObject);
begin
  SetWindowLong(Handle, gwl_ExStyle, GetWindowLong(Handle,
    gwl_ExStyle) or ws_Ex_Layered);
  TrackBar1Change (Sender);
end;
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  SetLayeredWindowAttributes(Handle, 0,
    255-TrackBar1.Position, lwa_Alpha);
end;
end.
```



➤ *Figure 1: Going, going, gone! If you want to create some snazzy visual effects that have your application window appearing and disappearing like the Marie Celeste, then Windows 2000 is undoubtedly the platform for you!*

disappears entirely. Beam me up Scottie!

There are a couple of important points to note here. Firstly, you will see from the screenshot that when a layered window becomes translucent, all of its child windows (read that as *controls*) become translucent as well. That's why I placed a pushbutton, a checkbox and a listbox on the form: their only purpose is to illustrate this point. If you think about it, this is exactly what we want to happen. Things would look pretty naff if we had a nice sexy translucent form, but the various controls on it remained resolutely opaque. Similarly, you will discover that pressing `Alt+PrintScreen` when a layered form is active will copy a standard, totally opaque image of the form to the Windows clipboard. This is because the internal screen-capture code within Windows references the off-screen, memory based bitmap of the window rather than the actual screen content after alpha blending.

Another thing you'll quickly discover is that once a window has been made completely invisible, it's gone for good! If you use the slider to make the demo program completely invisible, you *will* be able to move the slider back to a more central position, provided that you keep the mouse held down, which (of course) retains mouse capture. However, if you let go of the mouse while the slider is hard over to the right, you won't be able to 'find' the window again. This is because the windowing subsystem looks at the layered window, sees that it is completely invisible and passes all events straight through to the window behind.

Actually, in this particular case, it is possible to get the window back. A layered top-level window with zero opacity still shows up on the taskbar, and on the `Alt-Tab` switch windows list of running tasks. If you `Alt-Tab` to the window, and hit the `Home` key, the window will instantly reappear, assuming of course that the slider control still has the input focus!

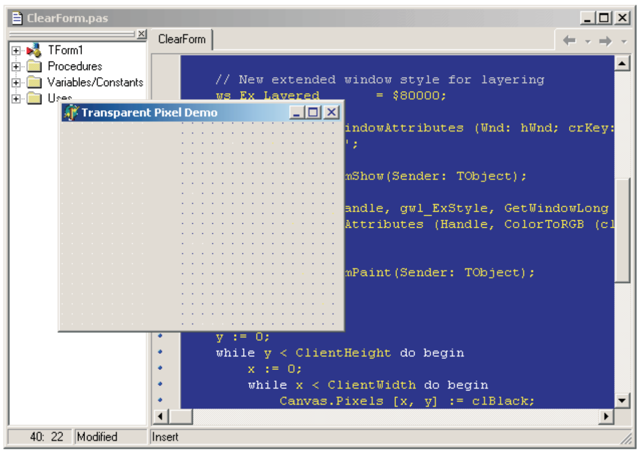
And Transparent Windows Too!

OK, so much for the `lwa_Alpha` key, but what about `lwa_ColorKey`? If you specify `lwa_ColorKey` as the final parameter to `SetLayered-WindowAttributes`, you can then supply a colour value in the `crKey` parameter. This basically tells Windows what colour you want to use for transparency. Any pixel in the window which matches the specified transparency colour will be considered transparent. It's important to understand that *this is done on a pixel by pixel basis.*

To see what I mean by this, take a look at the code fragment in Listing 4, for the sake of brevity, I haven't bothered to include the whole program this time. In this particular case, the `SetLayered-WindowAttributes` routine is passed the value `clBlack` as the colour key, with the final parameter set

➤ *Listing 4*

```
procedure TForm1.FormShow(Sender: TObject);
begin
  SetWindowLong (Handle, gwl_ExStyle, GetWindowLong (Handle, gwl_ExStyle) or
    ws_Ex_Layered);
  SetLayeredWindowAttributes (Handle, ColorToRGB (clBlack), 0, lwa_ColorKey);
end;
procedure TForm1.FormPaint(Sender: TObject);
var
  x, y: Integer;
begin
  y := 0;
  while y < ClientHeight do begin
    x := 0;
    while x < ClientWidth do begin
      Canvas.Pixels [x, y] := clBlack;
      Inc (x, 10);
    end;
    Inc (y, 10);
  end;
end;
```



```
// New extended window style for layering
ws_Ex_Layered        = $80000;
indowAttributes (Wnd: hWnd; crKey:
';
mShow(Sender: TObject);
andle, gwl_ExStyle, GetWindowLong
Attributes (Handle, ColorToRGB (cl
mPaint(Sender: TObject);
y := 0;
while y < ClientHeight do begin
    x := 0;
    while x < ClientWidth do begin
        Canvas.Pixels [x, y] := clBlack;
```

➤ *Figure 2: Look carefully, and you will see that this form contains a rectangular array of transparent pixels, somewhat like a sieve. As you move the form over the background, the sieve effect becomes very obvious, something that would be very difficult to achieve under Windows 9x without big performance penalties.*

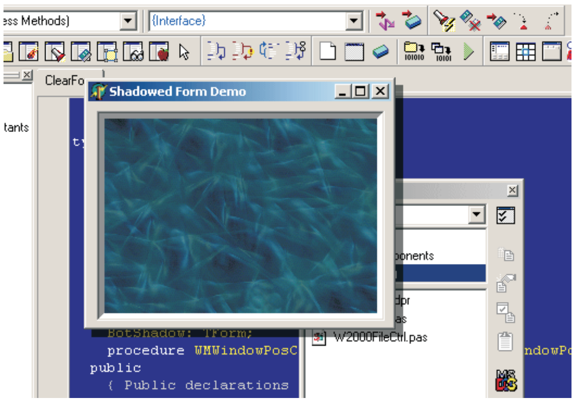
to `lwa_ColorKey`. This tells the operating system that every black pixel in the image should be transparent, and this is exactly what happens. I've added a custom `OnPaint` handler for the form which draws a grid of black pixels all over the surface of the form.

You can see the resulting effect in Figure 2. If you look carefully at this screenshot, you'll see that the pixel grid shows white dots where the window overlays Code Explorer, grey dots over the gutter area (which are invisible since the form itself is grey!) and dark blue dots over the code editor area. What's maybe less obvious is the effect on the caption bar. If you look *very* carefully at the minimise, maximise and close buttons on the right hand side of the caption bar, you'll notice that all the black pixels (the actual 'icons' on each button) are showing through as blue. In other words, transparency affects the whole window, not just the client area.

The same is true of hit testing. If I were to position the mouse so that its hot-spot exactly coincided with one of the transparent pixels on the form, then the cursor would change to reflect the cursor of the underlying window, eg an I-Beam when over the Delphi code editor. Similarly, clicking the mouse when it's exactly over a transparent pixel would bring the underlying window to the foreground.

`SetLayeredWindowAttributes` will also allow you to `OR` together the `lwa_Alpha` and `lwa_ColorKey` values, passing the result as the final parameter. In this way, you can set translucency and transparency characteristics for a single window in one call.

It's interesting to note that Windows 2000 already makes subtle use of alpha blending technology itself. If you look carefully at the mouse cursor, you'll see that it has a tasteful looking alpha blended shadow beneath it. As I mentioned in connection with `Alt+PrintScreen`, the Windows 2000 layering mechanism is invisible to most screen-capture code techniques. Consequently, if you use something like Paint Shop Pro and specify that you want to take a screenshot which includes the mouse cursor, you'll perhaps be surprised to discover a very boring plain-vanilla cursor as part of the image. Similarly, using Windows Explorer to drag shell folder items around looks much nicer under Windows 2000, because Microsoft use translucency to layer the drag image over the background image, creating a much more refined effect.



```
  BotShadow: TForm;
  procedure WMWindowPosC
public
  { Public declarations
```

➤ *Figure 3: And here's another cool effect, courtesy of the code in Listing 3. I've used two small translucent windows to create the effect of a real drop-shadow accompanying the top-level application window.*

Standing Out From The Crowd

OK Dave, this is all very well, but what can *I* actually *do* with this stuff? After all, not that many desktop programs are really going to need a translucent form, are they? Microsoft do give a few suggestions on what you might do with this new API feature, such as a hint/information window that pops up alongside your work, without obscuring what's underneath, but it's got to be admitted that opportunities for creating translucent top-level windows are fairly limited.

There's one rather nice effect we *can* apply to an application window, and that's to add a shadowed effect. Anyone who's spent more than five minutes with the Windows Help system will know that the Microsoft help engine incorporates the ability to display small popup windows that provide glossary information on unfamiliar terms. These windows appear to offer a sort of shadowed effect, but it's really a very cheap affair because the shadowing is done by dithering a grey brush with the underlying window content which doesn't give a particularly pleasing effect. Moreover, these popup windows don't move around. The whole thing is very modal, and the effect doesn't represent the use of a general purpose 'window shadowing mechanism' within the Windows API.

By making use of translucency effects in Windows 2000, we can achieve an effect that's a great deal better, as you can see in Figure 3. With a shadowed application window like this, you really can make your program stand out from the crowd! I'm not advocating that you should necessarily make all your application windows look like this, but if you have a form that's got something important to say, this sort of 'in your face' approach can sometimes be useful and, besides, you can easily change the width of the shadow area in order to vary the 'strength' of the effect.

So how did I manage this? By cheating, of course! The two shadow areas are made up of two additional windows that sit alongside the main application window. You can see the necessary code in Listing 5.

The `FormCreate` routine creates two new windows, `RightShadow` and `BotShadow` corresponding to the right and bottom shadow areas. Both of these forms are created with a `BorderStyle` of `bsNone` since we just want plain-vanilla rectangular areas without any caption bar or border. The shadowed form is set up as the owner of the two shadow areas, but it's important to set the `Parent` property `Application.MainForm`. If we set it to `Self` (ie the form for which we want to create a shadowed effect) then we'd obviously be unable to put the two shadowed areas outside of the form boundaries.

Things won't look too impressive if the shadow areas get left behind when the form is resized or moved, and consequently it's necessary to set up an `OnResize` event handler and also intercept the Windows `wm_WindowPosChanged` message which tells an application that a window position has been altered. Armed with this additional code, the shadow areas will obediently follow the form around, making a very nice effect. Notice that the `WMWindowPosChanged` handler checks that at least one of the shadow forms has been assigned before calling `FormResize`. This is necessary because Windows can potentially send a `wm_WindowPosChanged` message to a freshly created window, even before the `FormCreate` handler has been called. This would obviously cause the program to GPF if it started trying to access the two shadow forms before they'd been created.

You might also notice that the bottom shadow is drawn `ShadowWidth` pixels shorter than you might expect. This is necessary to prevent the two shadow areas from overlapping one another which would otherwise create a small box in the bottom-right corner with a different opacity to the rest of the shadow.

In this simple program, I've used a default shadow width of 7 and a shadow opacity of 150, but feel free to experiment with these values. Obviously, the higher the opacity, the darker will be the resulting shadow area. You might even fancy wrapping up the code into a reusable form class. One thing you really shouldn't change is the colour, `clBlack`, which I've used for the two shadow areas. I started off using various shades of grey, but found that although this looked good most of the time, it looked really unsavoury when 'shadowing' certain coloured backgrounds. I then tried `clWhite`, which looked even worse. Eventually, I realised that the whole thing about a shadow is that it's an absence of light, hence black! Using `clBlack` will give you a shadow that looks great irrespective of the background it moves over. Seems obvious in retrospect, but it wasn't at the time. Duh! You'll also find that using this technique will give you shadows with a much lower runtime overhead than other techniques because of the built-in layering support in Windows 2000.

Window Animation Techniques

Transparency and translucency effects are all very nice, but they're sort of static, aren't they? Wouldn't it be nice if Windows 2000 included support for animating things? Well, it does. The new platform includes a 'new' API routine called `AnimateWindow`.

If you're wondering why I put those quotes around 'new', let's not forget that this operating system has been in beta for several years. The `AnimateWindow` routine was first mentioned by Matt Pietrek back in 1997, when he announced that it would be included in the upcoming NT 5.0 (which turned into Windows 2000). Since NT5/W2K went into beta, Windows 98 has arrived and it also includes support for this call.

The presence of `AnimateWindow` shouldn't be too much of a surprise because, as I'm sure you know, Windows 2000 has those magical menus which fade in and out of reality. This is all done using `AnimateWindows`, the function prototype for which (translated from C/C++) is shown below:

```
function AnimateWindow(
  Wnd: hWnd; dwTime, dwFlags:
  DWord): Bool; stdcall;
  external 'user32.dll';
```

The routine is pretty straightforward to use; the first parameter is obviously a handle to the window which we wish to animate. The second parameter, `dwTime`, specifies the time (in milliseconds) during which an animation should take place. Microsoft have employed a *de facto* standard of 200 milliseconds for most of the animation effects in Windows 2000, which I'd say is about right. If you make the animation too long then power users will get cheesed off waiting, whereas if you make it too short, nobody will notice the effect, thus defeating the object of the exercise. Two hundred

➤ *Listing 5*

```
  ...
  private
    ShadowWidth: Integer;
    RightShadow: TForm;
    BotShadow: TForm;
  ...
procedure TForm1.FormCreate(Sender: TObject);
begin
  ShadowWidth := 7;
  RightShadow := TForm.Create (Self);
  RightShadow.Parent := Application.MainForm;
  RightShadow.BorderStyle := bsNone;
  RightShadow.Width := ShadowWidth;
  RightShadow.Color := clBlack;
  RightShadow.Visible := True;
  SetWindowLong (RightShadow.Handle, gwl_ExStyle,
    GetWindowLong (RightShadow.Handle, gwl_ExStyle)
    or ws_Ex_Layered);
  SetLayeredWindowAttributes(RightShadow.Handle, 0, 150,
    lwa_Alpha);
  BotShadow := TForm.Create (Self);
  BotShadow.Parent := Application.MainForm;
  BotShadow.BorderStyle := bsNone;
  BotShadow.Height := ShadowWidth;
  BotShadow.Color := clBlack;
  BotShadow.Visible := True;
  SetWindowLong (BotShadow.Handle, gwl_ExStyle,
    GetWindowLong (BotShadow.Handle, gwl_ExStyle)
    or ws_Ex_Layered);
  SetLayeredWindowAttributes(BotShadow.Handle, 0,
    150, lwa_Alpha);
  FormResize (Sender);
end;
procedure TForm1.WMWindowPosChanged(
  var Message: TWMWindowPosChanged);
begin
 Inherited;
 if Assigned (RightShadow) and RightShadow.Visible then
   FormResize(Nil);
end;
procedure TForm1.FormResize(Sender: TObject);
begin
  RightShadow.Height := Height;
  RightShadow.Left := Left + Width;
  RightShadow.Top := Top + ShadowWidth;
  BotShadow.Width := Width - ShadowWidth;
  BotShadow.Left := Left + ShadowWidth;
  BotShadow.Top := Top + Height;
end;
```

milliseconds seems a good compromise between the two.

The final, most important, parameter is dwFlags. This specifies the type of animation effect that we want. Possible values for dwFlags are shown in Table 1, and you can see the actual definitions for these constants in Listing 6. Where it makes sense, some of these values can be combined. For example, if AW_HOR_POSITIVE is combined with AW_VER_POSITIVE then you can make the window slide into view from its top-left corner.

➤ *Listing 6*

```
// AnimateWindow flags
aw_Hor_Positive      = $00000001;
aw_Hor_Negative      = $00000002;
aw_Ver_Positive      = $00000004;
aw_Ver_Negative      = $00000008;
aw_Center            = $00000010;
aw_Hide              = $00010000;
aw_Activate          = $00020000;
aw_Slide             = $00040000;
aw_Blend             = $00080000;
```

➤ *Table 1*

Value	**Description**
AW_SLIDE	Use slide animation. By default, roll animation is used. This flag is ignored when used with AW_CENTER.
AW_ACTIVATE	Activates the window. Do not use this value with AW_HIDE.
AW_BLEND	Uses a fade effect. This flag can be used only if the window is a top-level window.
AW_HIDE	Hides the window. By default, the window is shown.
AW_CENTER	Make the window expand outwards when showing, or collapse inwards when hiding (AW_HIDE).
AW_HOR_POSITIVE	Animate window from left to right. This flag can be used with roll or slide animation. It is ignored when used with AW_CENTER or AW_BLEND.
AW_HOR_NEGATIVE	Animate window from right to left. This flag can be used with roll or slide animation. It is ignored when used with AW_CENTER or AW_BLEND.
AW_VER_POSITIVE	Animate window from top to bottom. This flag can be used with roll or slide animation. It is ignored when used with AW_CENTER or AW_BLEND.
AW_VER_NEGATIVE	Animate window from bottom to top. This flag can be used with roll or slide animation. It is ignored when used with AW_CENTER or AW_BLEND.

It's important to note that AnimateWindow will only do the business when you're hiding or showing a window. It won't allow you to (for example) smoothly slide an existing, already visible, window from one location to another. Even so, it can be quite fun to play with.

If you're wondering what's the difference between a 'roll' and a 'slide', here's how it works. Let's take the case of a window appearing on screen. If you were to do a horizontal left to right slide, then the window would first appear as a thin vertical strip, expanding out to the left. As the width of the window increases, the window content slides towards the right. On the other hand, with a roll, the window content doesn't slide but is progressively revealed as the animation width increases. Think of a slide as the morning mail being shoved through the letterbox, and a roll as being like an unrolling carpet! See Listing 7.

As one would hope, the OnShow event handler is the most natural place to call AnimateWindow. This is demonstrated by the code snippet shown above. This will cause your application window to fade onto the screen. If you must, you can also use the same technique to fade out the application window when it's closed, see Listing 8.

I say 'if you must' because fading out tends not to work as well as fading in. Maybe Windows sends repaint requests to the application behind the disappearing window. In any event, the effect isn't as smooth, although I've found that you can partially compensate for this by increasing the fade out time to around 400 milliseconds as shown here.

There are a couple of other points to make here. Firstly, you will have difficulty using this technique if you have the form's Position property set to something other than poDesigned. The form will fade in to view and *then* suddenly snap to the required screen location as defined by Position! Needless to say, this doesn't look too great! If you look at the code in FORMS.PAS, you will see that this happens because the VCL

```
procedure TForm1.FormShow(Sender: TObject);
begin
  AnimateWindow (Handle, 200, aw_Blend);
end;
```

➤ *Above: Listing 7*

➤ *Below: Listing 8*

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  AnimateWindow (Handle, 400, aw_Blend or aw_Hide);
end;
```

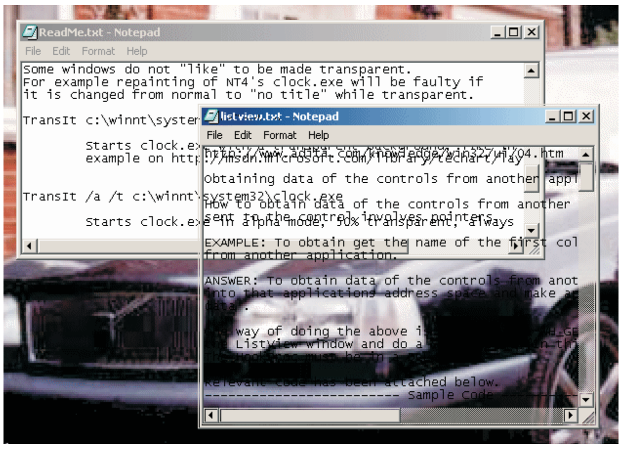
library calls the OnShow handler before actually moving the window to its designated position. There are probably ways around this, but...

The second thing you need to be aware of is that well-behaved Windows 2000 applications need to take note of the user's preferences regarding animation effects. You can get this information by passing various new parameters to the SystemParametersInfo API routine. The most important query parameters are:

```
SPI_GetMenuAnimation = $1002;
SPI_GetMenuFade      = $1012;
```

Firstly, you use the SPI_GetMenuAnimation parameter to determine whether or not animated menus are enabled. If so, then you can use the SPI_GetMenuFade parameter to figure out whether the user prefers sliding menus, or menus that fade in and out. Strictly speaking, these parameters apply to menus, but the chances are that if the user doesn't want animated menus, then he/she won't appreciate your animated application windows either!

That said, there are some circumstances where it is acceptable to display animation effects irrespective of the user's personal preferences. If you want to give your application a fancy About box, for example, then I reckon it's perfectly OK to pull out all the stops in this particular case! There are also situations where animation might constitute an integral part of your user interface, such as those delightful sliding



```
Some windows do not "like" to be made transparent.
For example repainting of NT4's clock.exe will be faulty if
it is changed from normal to "no title" while transparent.

TransIt c:\winnt\system32\clock.exe

        Starts clock.exe
        example on http

TransIt /a /t c:\winnt\system32\clock.exe

        Starts clock.exe in alpha mode, 50% transparent, always
```

```
http://www.adita.com/knowledge/win32/ui/04.htm
://msdn.microsoft.com/library/techart/lay

Obtaining data of the controls from another appl

How to obtain data of the controls from another
sent to the control involves pointers

EXAMPLE: To obtain get the name of the first col
from another application.

ANSWER: To obtain data of the controls from anot
into that applications address space and make a
data.

One way of doing the above is
the ListView window and do a
The HookProc must be in a

Relevant code has been attached below.
------------------------ Sample Code --------
```

➤ *Figure 4: Notepad 'Classic Edition' meets its see-through sibling! Using the TRANSIT utility (referred to in the text) you can make any Windows application exhibit transparency/translucency effects without changing the executable in any way. Clever, huh?*

drawers used by Kai's Photo Soap from MetaTools, Inc. The same could be said of custom menu implementations.

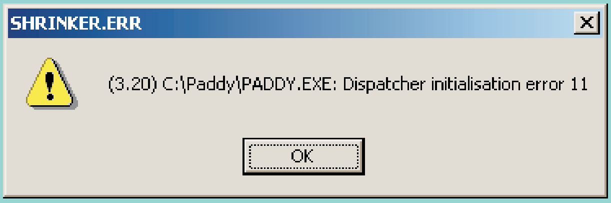
On the subject of pull-out drawers and other similar effects, Windows 2000 introduces a new API call, UpdateLayeredWindow which complements the functionality of the aforementioned SetLayeredWindowAttributes routine. This routine allows you to specify things such as window size and position on the fly, together with a device context handle which represents the current drawing surface of the window. Allegedly, this routine allows you to create effects that are even more powerful than those I've described so far, but quite frankly the Microsoft documentation is very poor and I haven't been able to find any decent examples of how to use this call in anger in my investigations so far.

Shrinker: Don’t Call Us, We’ll Call You

As you are no doubt aware, the practice of compressing executables has become very common in recent years. A compressed file (be it an EXE file, a DLL, an OCX or whatever) works just like an ordinary non-shrunk executable, but it is substantially smaller and more difficult for hackers to disassemble and patch. For both of these reasons, many people now routinely use Shrinker, ASPack, Fusion and a variety of other EXE compression tools. Allaire HomeSite and TurboPower’s Sleuth QA Suite (both written in Delphi, interestingly enough) are just a couple of examples of products which have been protected from prying eyes through the use of EXE compressor technology.

Some time ago, I used Delphi to write a reader survey program for another computer magazine and I was surprised to receive a bug report to the effect that a small number of end users couldn’t even start the provided EXE file. After some investigation, I discovered that this was due to an operating system incompatibility with the EXE compressor I was using, Shrinker from Blink Inc (whose website is at www.blinkinc.com). All compressed EXE files have a small stub loader, a chunk of code which is called by the operating system and is responsible for decompressing the rest of the file into memory. It turned out that Shrinker’s stub loader was blowing up under some circumstances. I provided an uncompressed version of my survey program to the magazine in question, and the problem did not recur...

...Until, that is, I upgraded my main development system to Windows 2000. I have a small desktop utility, Paddy, which I wrote for my own use. It helps to organize my work and remember what I’m supposed to be doing from one day to the next. Think of it as a sort of memo-pad application. This little utility worked fine until it came across Windows 2000, at which point I got the result shown in Figure 5. Let me stress that this isn’t the fault of Windows 2000. No, it’s good old Shrinker screwing up again. It turns out that I’d used Shrinker to compress Paddy and, yes, you’ve guessed, the Shrinker 3.2 loader doesn’t like Windows 2000. Yes, I know Blink Inc are now up to version 3.4, but so what. This isn’t the first time I’ve been bitten in the backside by Shrinker but it’ll certainly be the last. If you’ve been using Shrinker to compress your deployed executables, then I’d suggest that you just might want to look elsewhere for a solution that will be more robust *[And of course we’d love to hear from Blink Inc with their comments... Ed]*.


SHRINKER.ERR

(3.20) C:\Paddy\PADDY.EXE: Dispatcher initialisation error 11

➤ *Figure 5: The less said about certain EXE compressors, the better. Sadly, many out-in-the-field applications which have been compressed using quite recent versions of Shrinker won't run under Windows 2000, through no fault of the new platform itself.*

Next Month

Be that as it may, I hope to dig up some more information for next month’s column, when I’ll be continuing this discussion of the enhanced visual effects that are possible under Windows 2000. I’ll be looking at some other API routines, and we’ll be concentrating on how to use all these goodies to enhance existing Delphi controls, this is where the fun *really* starts!

In the meantime, if you point your browser at http://gallery.uunet.be/lucvdveken, you’ll find an interesting little command-line utility called TRANSIT (go to the Windows 2000 download area and grab the file called TRANSIT.ZIP) which allows you to add translucency and transparency effects to any existing application at start-up time. See Figure 4 for an... umm... *interesting* example of what can be done.

If you look at the two sample projects included on this month’s disk, you’ll see that I’ve had to add the necessary API call declarations, constant values, etc, into the program source code as for example in Listings 1 and 4. This is because the necessary Windows 2000 declarations haven’t yet made it into Borland’s RTL units such as WINDOWS.PAS. Surprisingly, even the recently released Delphi 5 Update CD (obtainable for the princely sum of a tenner from Borland UK HQ) doesn’t remedy this situation. You will, however, find the necessary SDK declarations in the header files accompanying C++Builder 5, but of course, they have to be translated into the equivalent Delphi syntax. Delphi 5 does include the declaration for the aforementioned `AnimateWindow` routine, but so it should for a call that’s been documented since 1997!

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is Technical Editor of *Developers Review* which is also published by iTec. You can contact Dave at TechEditor@itecuk.com