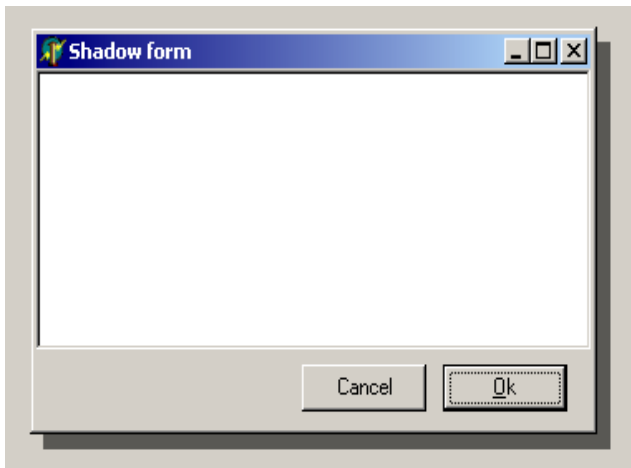


Shadow Component

by Dave Oldfield

During a recent project, my client found the application difficult to use because the modal forms were becoming lost in the background as they were both silver coloured. I tried changing the colour of the forms but the client was still not happy with the effect. Then I remembered an article written by Dave Jewell in the Delphi Magazine, issue 57, where he showed how to use a new (Windows 2000) API call to make translucent forms. He then went on to show how the effect could be used to attach shadows to a form and I felt this was just what I needed to give the modal forms that extra bit of depth. Rather than building the code into the form hierarchy and using inheritance, I decided to turn the code into a component, which could be more easily used elsewhere. The component illustrates some useful techniques and I thought it would make an interesting article for the magazine.

An example of the shadow effect can be seen in the Delphi help. Clicking on the 'Hierarchy' label for a component pops up a window with attached shadows and the text behind can be seen through them. Here, the shadow effect is done with a 'spotty' brush: some pixels are left transparent and the background shows through. The new API call uses proper *alpha blending* where the foreground and background are merged together. The effect looks good, much better than a dithered brush, and is very fast too - see diagram below.



The shadows are added to the form by creating some small additional windows that are drawn next to the target form. These additional forms are created with the new API call so that they are blended with the background. To do this, the parent of a shadow form must be set to nil - Delphi then uses the desktop as the parent and the shadows will merge with whatever is behind them, whether part of your application or not.

If the owner form is moved or resized, then the shadows must be redrawn too. To track the changes the component must intercept some of the form's events and / or window messages. At first I thought of intercepting the Resize event

but this is not called if the form is moved. However, the WM_WINDOWPOSCHANGED message is called when the size or position is changed - just what is needed. So for the component to intercept this message it must provide an alternative procedure somehow. The best candidate is the target form's WindowProc procedure: by subclassing it, and providing an alternative at runtime, the component can track all changes to the size and position of the form.

In the code listing which follows, notice the extra API call, SetLayeredWindowAttributes, which is used to set the attributes required for alpha blending. The call is not included in any of Delphi's units and so must be added manually. You may want to put it into a separate unit rather than in the component unit, as here, because the next version of Delphi will probably include it. For full details of this function please refer to the Delphi Magazine article mentioned earlier.

The TShadower component only has one property, Enabled, which does exactly as you might expect - by default it is True. Several things happen in the component's constructor. As we need to attach the shadows to the owner form, a reference to it is stored in the FOwner variable and, as we know this will be a form, it is typecast as that. After setting the Enabled property's default value the owner's window procedure is intercepted. A reference to the existing WindowProc method is stored in the FOldWindowProc variable and we then supply our own version.

Next the two shadow forms are created with the CreateNew method. As can be seen from the code listing, the shadow forms are declared as a separate class called TShadow. Although the TShadow class is a TForm descendant, no DFM file is declared. To use the normal Create constructor we need a DFM file - Delphi will complain without it - but it is unnecessary in this case as Delphi knows how to stream standard forms (with no components) without one. The CreateNew method is overridden to provide the default settings for the shadow forms - no border, black colour, etc. Also, it is here that the alpha blending is set up with the call to the SetLayeredWindowAttributes method. Notice we test that we are not in design mode as there is no point in doing any of this in the IDE.

The rest of the component code is very straightforward. In the new WindowProc method, the form's existing method **must** be called first to allow Windows to do whatever it needs to do - omit this and the window will not appear at all. Then the message type is tested and if it is WM_WINDOWPOSCHANGED the DrawShadows procedure is called, it also needs to be called when the Enabled property is changed. All the DrawShadows procedure does is set the BoundsRectangle for each of the shadow windows, depending on the position and size of the target form.

To save space in this article I have simplified the component that I actually use in practice. You could add properties to modify the width and darkness of the shadows. The width has been hard-coded as 8 pixels in the DrawShadows procedure and the darkness is the value of 150 in the SetLayeredWindowAttributes call (the darkness can vary between 0 and 255 – higher is darker). Purists may like to leave the creation of the shadows until they are actually used - easy to add code to the DrawShadows procedure and create them if unassigned. The component only works with Windows 2000 and you will need to test for this if using in mixed environments. Finally, if using it for modeless forms, code will have to be written to hide the shadows on closing of the form.

Although, I have shown a component that has a specific job, the technique of using components to intercept events and messages on a form can be easily extended. You could have a library of components each of which modifies a form in some special way - the form's constructor can be intercepted just as easily. By encapsulating into a component, the behaviour is easily maintained and reused. Enjoy.

```

type
  TShadow = class(TForm)
  public
    constructor CreateNew(aOwner: TComponent;
      Dummy: Integer = 0); override;
  end;

  TShadower = class(TComponent)
  private
    FOwner: TForm;
    FOldWindowProc: TWndMethod;
    FEnabled: Boolean;
    FBottomShadow: TShadow;
    FRightShadow: TShadow;
    procedure SetEnabled(Value: Boolean);
    procedure WindowProc(var Message:
      TMessage);
  public
    procedure DrawShadows;
  public
    constructor Create(AOwner: TComponent);
      override;
    destructor Destroy; override;
  published
    property Enabled: Boolean read FEnabled
      write SetEnabled default True;
  end;

  procedure Register;

implementation

{ Windows 2000 }

const
  ws_Ex_Layered = $80000;

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

{ TShadow }

constructor TShadow.CreateNew(aOwner:
  TComponent; Dummy: Integer = 0);
begin
  inherited;
  Parent := nil;
  BorderStyle := bsNone;
  FormStyle := fsStayOnTop;
  Color := clBlack;

```

```

  SetWindowLong(Handle, gwl_ExStyle,
    GetWindowLong(Handle, gwl_ExStyle) or
      ws_Ex_Layered);
  SetLayeredWindowAttributes(Handle, 0, 150,2);
end;

{ TShadower }

constructor TShadower.Create(AOwner:
  TComponent);
begin
  inherited;
  FOwner := AOwner as TForm;
  FEnabled := True;
  if not(csDesigning in ComponentState) then
  begin
    FOldWindowProc := FOwner.WindowProc;
    FOwner.WindowProc := Self.WindowProc;

    FRightShadow := TShadow.CreateNew(Self);
    FBottomShadow := TShadow.CreateNew(Self);
  end;
end;

destructor TShadower.Destroy;
begin
  { just in case.. }
  if Assigned(FOldWindowProc) then
    FOwner.WindowProc := FOldWindowProc;
  inherited;
end;

procedure TShadower.WindowProc(var Message:
  TMessage);
begin
  {Call original message first..}
  FOldWindowProc(Message);
  {..then only this one..}
  if (Message.Msg = WM_WINDOWPOSCHANGED) then
    DrawShadows;
end;

procedure TShadower.SetEnabled(Value: Boolean);
begin
  FEnabled := Value;
  if not(csDesigning in ComponentState) then
    DrawShadows;
end;

procedure TShadower.DrawShadows;
begin
  FRightShadow .SetBounds(
    FOwner.Left + FOwner.Width,
    FOwner.Top + 8, 8,
    FOwner.Height);
  FRightShadow.Visible := Self.Enabled;

  FBottomShadow.SetBounds(
    FOwner.Left + 8,
    FOwner.Top + FOwner.Height,
    FOwner.Width - 8, 8);
  FBottomShadow.Visible := Self.Enabled;
end;

procedure Register;
begin
  RegisterComponents('Samples', [TShadower]);
end;

```

Dave Oldfield is a Delphi/InterBase consultant and developer, interested in contract work. He is based in Sheffield and London, and is a regular participant at BUG meetings. You can contact him on 0114 235 6571 or dave@ifersys.demon.co.uk.