Beating The System: Groovy Group Boxes

by Dave Jewell

So what's with this group box stuff, Dave? At the end of last month's column, you promised to do some more with Windows 2000 transparency and translucency effects, applying them to individual controls rather than the application window. So where is it, eh? Well, I was afraid you were going to ask that! The fact is that... err... it didn't work. All that wonderful window layering I discussed last month only seems to work for top level windows. So there you have it, or not as the case may be.

Building Better Group Boxes

This month, I'm going to tiptoe quietly away from translucency and take a look at how to improve Delphi's group box component. My original plan was to produce some really dazzling new Delphi controls, courtesy of the translucency support in Windows 2000, but since that's not worked out, I'm hoping that this month's group box control will placate you instead!

This article was prompted by some recent consultancy on a large commercial project. The developers were keen to trim some fat from their seriously bloated application and wanted advice on how to do it. Having looked through their source, it quickly became obvious that one of their misdemeanours was repetition of very similar code across many different forms. This was done for a variety of reasons, one being that they wanted to enforce a consistent user interface throughout the application.

As seasoned Delphi developers will know, the standard group box has many deficiencies, perhaps the main one being that a disabled group box doesn't automatically disable all the child controls contained within it. For sure, it's impossible for the end-user to actually interact with the controls inside a disabled group box, but nevertheless these controls continue to look very un-disabled! Worse, Borland's group box does not even bother to make itself look disabled, so the net effect (from a naive end-user's viewpoint) is what looks like an enabled group box containing a set of what look like enabled controls, all of which stubbornly ignore any attempt at user interaction. Great confusion and many tech support calls are the inevitable results!

In an effort to eliminate this confusion, the aforementioned developers decided to write code which explicitly disabled each and every child control when a group box was disabled, enabling those controls again when the group box was enabled. This can obviously amount to a lot of tedious, repetitious code, but they decided to go further by changing the `Color` property of various controls to reflect their disabled state.

To understand what I mean here, consider the edit box, listbox, combobox, memo control and date-time picker. All these components have a white (strictly speaking, it's `clWindow`, but it's white with the standard Windows colour scheme) 'type here!' area which remains resolutely white even when the control is disabled. The folks I spoke to thought this was confusing to inexperienced Windows users (I tend to agree), and they wanted the `Color` property to automatically change to `clBtnFace` when the control was disabled.

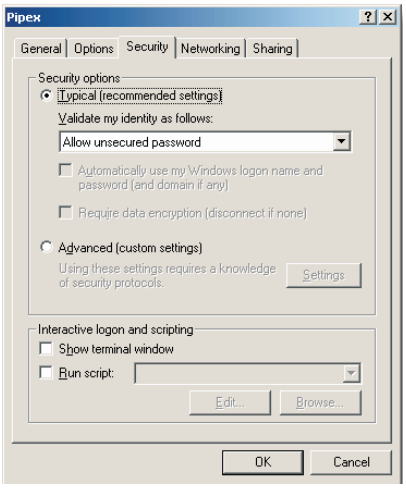
What these developers had done was to add reams of explicit code which altered the colour and enabled/disabled state of each child control whenever the enclosing group box changed from enabled to disabled, or vice versa. All this 'grunt-level' code had to be replicated for each group box on the form, and for each form in the application. I gently suggested that by encapsulating the required functionality into a custom version of the group box component, they could dispense with a huge amount of messy code which made it more difficult to see what each source module was really trying to achieve. Thus was born the idea of a new, improved, group box.

But maybe you've never used the technique of enabling and disabling group boxes in your own application? If not, consider the sort of situation shown in Figure 1. This dialog is part of Microsoft's RAS support. The top of the dialog contains a `Security options` group box which is subdivided into two other options, `Typical` and `Advanced`. When `Typical` is selected, the `Advanced` items (a text label and pushbutton) are shown as disabled, whereas selecting `Advanced` disables all the controls in the `Typical` area: the two checkboxes, combobox, and the label control above the combo.

In this situation, there's only one group box, the `Security options` box that encloses everything. But imagine how much handier things would be if we had a group box around each of the 'sub-groups' that I've just described. If these group boxes *really did* disable their children properly, then it would only be necessary to enable/disable a single component rather than several. And as I always say, less code equals less bugs. But how can we put a group box around a set of controls without the group box borders showing up? With an ordinary group box you can't, of course, but with my group box, you can do just that and a lot more besides. But I'm getting ahead of myself, let me introduce you to `TGroupBoxEx`.

What's New In TGroupBoxEx

You might be forgiven for thinking that there's not much about the humble group box that can be improved but I hope to persuade you otherwise. Let's begin with something extremely trivial, but which irritates my concept of artistic perfection. If you look closely at



➤ *Figure 1: Here's a Microsoft dialog which would benefit from decent group boxes. Internally, this RAS configuration dialog is undoubtedly complicated by the need to explicitly enable/disable each control according to the state of the two 'master' radio buttons.*

the caption string of an ordinary group box, you'll see that the horizontal group box border butts right up against it on either side, nasty. I don't like returning to my car to find other folks have parked their cars half an inch from my front and rear bumpers, and I feel that the group box caption string needs a bit more breathing space too. I know I'm not the only Delphi/C++Builder developer who feels the same way, because I've noticed other programmers add a space to the front and back of their group box caption string. `TGroupBoxEx` will do the job for you, automatically, via a new `Boolean` property called `CaptionSpaces`.

I've already mentioned that I wanted to be able to inhibit the display of the group box border. This is accomplished through another `Boolean` property called `ShowBorder` which does exactly what it says on the tin. In fact, if you set `ShowBorder` to `False` and specify an empty caption string you've effectively got yourself a 'stealth' group box. From the end-user's perspective, he/she doesn't see a group box at all, but from a programming perspective you can still use the group box as a handy container for multiple components and, as I've pointed out already, you can conveniently use a stealth group box as a way of enabling/disabling a whole slew of related components via a single assignment to the group box `Enabled` property.

This brings me neatly on to one of the most important features of `TGroupBoxEx`: the ability to automatically enable/disable child components. In the foregoing discussion, I've already pointed out that not only is this a great idea from the perspective of reducing repetitive code, but it also provides better visual cues to the user, especially if you follow through on my idea of greying-out edit boxes, comboboxes, and so forth. Again, `TGroupBoxEx` will do this automatically for you, because it recognises the following five child component classes and sets each child component's `Color` property to either `clBtnFace` or `clWindows` according to whether or not the group box is being disabled: `TEdit`, `TListBox`, `TComboBox`, `TMemo` and `TDateTimePicker`.

Now I know what you're thinking: what if you don't want to make use of this automatic greying-out facility? Well, OK, if you're happy having edit boxes, memo boxes, etc, that are disabled but don't actually look disabled, then I won't stop you. I won't encourage you, but I won't stop you either! To cater for this scenario, `TGroupBoxEx` provides a custom event called `TGroupBox- EnableDisableQuery` which looks like this:

```
TGroupBoxEnableDisableQuery =
  procedure (Sender: TObject;
  Control: TControl;
  Enabled: Boolean; var
  Handled: Boolean) of Object;
```

When a `TGroupBoxEx` component is being enabled or disabled, the group box will call the above event handler for every child control in the group box. If you don't want to take advantage of the automatic greying-out facility, you can simply set the `Handled` parameter to `True`, that's all you have to do. In such a case, all the disabled child controls will continue to look enabled, even though they're not. If you leave the `Handled` parameter set to its default value of `False`, then the group box will go ahead and perform automatic greying-out for you.

But this raises another question; what if you *do* want to take advantage of the auto-greying facility, but you have some special custom control class that's not recognised by `TGroupBoxEx`? Or maybe you want all your disabled `TEdit` boxes to appear as fluorescent pink irrespective of the current Windows colour scheme. Again, this is up to you. The above function prototype takes four parameters. `Sender` is the source of the event and therefore corresponds to the `TGroupBoxEx` control itself. `Control` is a reference to the child control that's being enabled or disabled, and of course, you can use `TObject.ClassName` to determine the type of control in the usual way. The `Enabled` parameter shows whether the control is being enabled or disabled, and we've already discussed the `Handled` parameter.

To put this in concrete terms, suppose you've implemented a group box which contains a `TSpinEdit` control, which isn't one of the 'standard' classes recognised by `TGroupBoxEx`. In order to perform automatic greying-out of all spin-edit controls within all `TGroupBoxEx` boxes on your form, just point all the group boxes at a shared `OnGroupBoxEnableDisableQuery` event handler which looks like Listing 1. As you can see, it specifically checks for `TSpinEdit` and

➤ *Listing 1*

```
procedure TForm1.GroupBoxEx1EnableDisableQuery(Sender: TObject; Control:
  TControl; Enabled: Boolean; var Handled: Boolean);
begin
  if Control.ClassName = 'TSpinEdit' then begin
    if Enabled then
      TSpinEdit (Control).Color := clWindow
    else
      TSpinEdit (Control).Color := clBtnFace;
    Handled := True;
  end;
end;
```

does the necessary according to the state of the Enabled parameter.

This raises one final question. Why didn't I just make TGroupBoxEx automatically recognise TSpinEdit and every other 'greyable' VCL component that's going? The answer is simple: for that to work, you'd effectively have to link all those infrequently-used control classes into any application that uses TGroupBoxEx, irrespective of whether or not the application ever used those control types. I felt that concentrating on the five aforementioned control classes represented a good compromise.

While we're on the subject of enabling/disabling the group box, I've also written TGroupBoxEx so that when the group box is disabled, the group box caption string is correctly drawn with a sunken, greyed-out, appearance, unlike the standard issue group box.

Another feature of TGroupBoxEx is the ability to display the caption string in one of twelve possible positions. For each of the four sides of the group box itself, the caption bar can appear at either end or in the middle. This is controlled through another new enumerated type property, CaptionPos (see Listing 2).

If you're wondering what the difference is between gbBottomRight and gbRightBottom, the naming convention that I've adopted is to specify the side of the box first, followed by the position along that side. In other words, gbBottomRight will locate the caption string on the bottom side of the group box, with the string in the rightmost position. gbRightBottom will place the caption string on the right hand side of the box, with the string in the bottom position, and so on. When the caption string is located on the left or right sides of the group box, the font that's used will automatically be rotated through either 90 or 270 degrees so that it still sits vertically alongside the sides of the group box. This facility only works with TrueType fonts.

Group boxes being what they are, developers sometimes want a caption string that's far more noticeable than the standard issue. The big problem here is that if you select a nice 'fat' font for your group box caption, all the controls inside your group box will adopt the same look because, as I'm sure you appreciate, Delphi controls have their ParentFont properties set True by default. This means that you're forced to set the ParentFont properties of all the child controls to False and that in turn makes things more tedious when you want to apply font changes that affect them all. To get around this, TGroupBoxEx introduces a new TFont property called CaptionFont, which affects only the the caption.

The last (but by no means least!) TGroupBoxEx enhancement is the ability to render a custom background behind the caption. I particularly wanted to add this feature to my group box control because I've seen some other shareware group box controls that allow you to put a graduated fill, bitmap or whatever behind the group box caption. Used tastefully, and in conjunction with the CaptionPos and ShowBorder properties, you can get some nice effects. The custom caption background facility is implemented via another event handler which looks like this:

```
TGroupBoxPaintCaptionBackground
  = procedure (Sender: TObject;
  Canvas: TCanvas; const Rect:
  TRect; var Handled: Boolean)
  of Object;
```

As expected, Sender is the group box control that's requesting the custom caption background paint. Canvas is the canvas to use for painting the custom background, whereas Rect represents the area available for painting. There are a couple of points to make here. Firstly, if you want to write a generalised graduated fill handler or something of that nature, you need to remember that the group box can handle vertically oriented captions. Thus, to determine the graduated fill direction you could cast Sender to TGroupBoxEx and check the CaptionPos property or, more simply, just examine the passed rectangle to determine which is the longest side and thus the orientation. In most cases, this isn't likely to be needed because, of course, you already know exactly how you've set up a particular group box. Such considerations only apply if you're using a shared OnPaintCaptionBackground event handler with multiple group boxes on the same form, some of which have a vertically oriented caption, and some don't. Needless to say, this wouldn't represent a very appealing user interface!

Secondly, bear in mind that the passed Rect argument defines the prospective caption area along an entire side of the group box. In other words, the rectangle isn't 'auto-sized' according to the length of the caption string, but it is auto-sized according to the width or height of the group box. I did things this way because I like having a nice graduated fill that covers most of the width/height of the group box. However, since you can easily access the Canvas and Caption properties via the OnPaintCaptionBackground handler, it's no big deal to call TCanvas.TextWidth and calculate a bounding rectangle that closely hugs the text.

How It All Works

OK, enough of the new features list. You can see the full source for TListBoxEx in Listing 3. Let's take a walk through the code listing. The first point to note is that, rather than bunging all the aforementioned new properties directly into the control, I chose to place them inside a TPersistent class called TGroupBoxOptions. This class, in turn, is exposed by the group box control itself as a property called Advanced. As experienced component developers will realise, this has the effect of turning all those new properties into nested properties, each of which appears

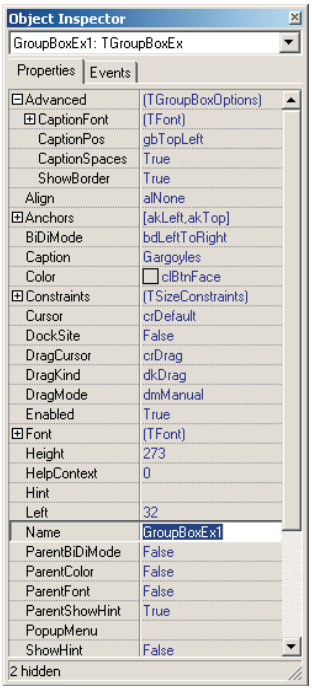
➤ *Listing 2*

```
TGroupBoxCaptionPos = ( gbTopLeft, gbTopMiddle, gbTopRight, gbBottomLeft,
  gbBottomMiddle, gbBottomRight, gbLeftTop, gbLeftMiddle, gbLeftBottom,
  gbRightTop, gbRightMiddle, gbRightBottom );
```

inside the Advanced property. You can see this happening in Figure 2.

This approach requires a little more work, but it is worthwhile, especially when you're effectively supplementing the functionality of an existing component. By grouping all the new properties under a single Advanced property, experienced developers can quickly find the new features. Moreover, by using a property name of Advanced, the Object Inspector will sort the property list such that this property generally appears first, making it even easier to find.

The TGroupBoxOptions.Create constructor is called from the group box constructor, creating the Options object and setting the default parameter values. At the same time, it sets up an OnChange handler for the caption font object so that the control is automatically notified whenever the caption font gets changed. This gets routed to the CaptionFontChanged method, which in turn calls the Changed procedure to notify the group box that it needs to redraw itself. The same mechanism is invoked when the CaptionSpaces property is changed, CaptionPos is modified, and so on. As I pointed out, this does add a little to the complexity of the code, but not much, and I think it's well worth it for the convenience of having all the new properties neatly grouped together.



➤ *Figure 2: When enhancing an existing component, it's often good to nest new properties into an 'Advanced' property, so it is immediately obvious what functionality is being added.*

Yes, I realise that Delphi 5 introduces property categories within the Object Inspector which represents an alternative approach to traditional nested properties. However, aside from the obvious point that this only works with Delphi 5 and C++Builder 5, I don't particularly like the way Borland implemented this. As I said when we reviewed Delphi 5 in *Developers Review*, I would have preferred to see a mechanism which allowed the component user to freely create, delete and rename categories, moving items from one category to another at will.

The heart of TGroupBoxEx is TCustomGroupBoxEx, which does all the real work. It exists so you can derive your own variations from this class, choosing not to publish certain properties as appropriate. The TCustomGroupBoxEx, in turn, is derived from TCustomGroupBox. You might be forgiven for thinking that the latter works by sub-classing Microsoft's underlying API-level control, but interestingly it doesn't seem to: it's a pure VCL control.

The fun starts in the TCustomGroupBoxEx.CMEnabledChanged routine which takes care of walking through the list of child controls, enabling or disabling each one as required. You'll notice the call to Invalidate here which is obviously needed so that the group box will update its own appearance too. For each child control in the group box, the routine checks to see if the fOnEnableDisableQuery event handler has been assigned. If so, it's called to give the application an opportunity to make custom changes to the appearance of the control, as discussed earlier. If the application sets the Handled flag, then CMEnabledChanged simply skips the child control and moves on to the next one. Notice that setting this flag to True means that the group box doesn't even attempt to set the Enabled property of the control, the assumption is that if the

➤ *Facing page, Listing 3*

application wants special processing, then it knows what it's doing. Once we've got the go-ahead to enable/disable the child controls, and auto-grey their appearance (as discussed earlier), the code just checks for the five recognised control classes and does the business as necessary.

The Paint method is responsible for drawing the actual group box border and for calling the PaintCaption routine. Parts of this code bear some resemblance to the original Paint method in TCustomGroupBox! Needless to say, what goes on here isn't too tricky, the main thing is to indent one of the four sides of the group box according to wherever the caption is currently positioned. This allows for the caption string to straddle the group box border in the usual way. Notice that I use Canvas.TextHeight to determine the height of the caption string. This obviously becomes the 'width' (in physical terms) if the caption is displayed in vertical orientation.

Back in the old days, this wouldn't have worked because monitors didn't have square pixels or, to put it another way, the horizontal pixels per inch resolution was different to the vertical resolution. This meant that if you rotated a font through 90 degrees, the 'width' of a vertical font would be different to the height of the corresponding horizontal font. Ever since the advent of VGA, Windows has been blessed with square pixels which certainly simplifies code such as that shown here.

If the Caption property is a non-empty string, then PaintCaption is called to draw it, and this is where most of the tricky stuff takes place. Like most folks, I like to do the easy bits first, so PaintCaption handles the six possible horizontal text positions first. After padding the caption string with spaces according to the setting of the CaptionSpaces property, the code then determines the pixel width and height of the resulting string. A notional bounding rectangle is

```
unit GroupBoxEx;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls,Forms, Dialogs, StdCtrls;
type
  TGroupBoxCaptionPos = ( gbTopLeft, gbTopMiddle,
    gbTopRight, gbBottomLeft, gbBottomMiddle, gbBottomRight,
    gbLeftTop, gbLeftMiddle, gbLeftBottom,
    gbRightTop, gbRightMiddle, gbRightBottom );
  // Custom procedures
  TGroupBoxEnableDisableQuery = procedure (Sender: TObject;
    Control: TControl; Enabled: Boolean; var Handled:
    Boolean) of Object;
  TGroupBoxPaintCaptionBackground = procedure (Sender:
    TObject; Canvas: TCanvas; const Rect: TRect; var
    Handled: Boolean) of Object;
  TGroupBoxOptions = class (TPersistent)
  private
    fOnChange: TNotifyEvent;
    fCaptionSpaces: Boolean;
    fShowBorder: Boolean;
    fCaptionFont: TFont;
    fCaptionPos: TGroupBoxCaptionPos;
    procedure Changed;
    procedure CaptionFontChanged (Sender: TObject);
    procedure SetShowBorder (Value: Boolean);
    procedure SetCaptionPos (Value: TGroupBoxCaptionPos);
    procedure SetCaptionFont (Value: TFont);
    procedure SetCaptionSpaces (Value: Boolean);
  public
    constructor Create;
    destructor Destroy; override;
  published
    property OnChange: TNotifyEvent read fOnChange
      write fOnChange;
    property CaptionSpaces: Boolean read fCaptionSpaces
      write SetCaptionSpaces default True;
    property ShowBorder: Boolean read fShowBorder
      write SetShowBorder default True;
    property CaptionPos: TGroupBoxCaptionPos
      read fCaptionPos
      write SetCaptionPos default gbTopLeft;
    property CaptionFont: TFont read fCaptionFont
      write SetCaptionFont;
  end;
  TCustomGroupBoxEx = class (TCustomGroupBox)
  private
    fOptions: TGroupBoxOptions;
    fOnEnableDisableQuery: TGroupBoxEnableDisableQuery;
    fOnGroupBoxPaintCaptionBackground:
      TGroupBoxPaintCaptionBackground;
    procedure OptionsChanged (Sender: TObject);
    procedure CMEnabledChanged (var Msg: TMessage); message
      cm_EnabledChanged;
  protected
    procedure AdjustClientRect (var Rect: TRect); override;
    procedure Paint; override;
    procedure PaintCaption (Str: String);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Advanced: TGroupBoxOptions read fOptions
      write fOptions;
    property OnEnableDisableQuery:
      TGroupBoxEnableDisableQuery read fOnEnableDisableQuery
      write fOnEnableDisableQuery;
    property OnPaintCaptionBackground:
      TGroupBoxPaintCaptionBackground
      read fOnGroupBoxPaintCaptionBackground
      write fOnGroupBoxPaintCaptionBackground;
  end;
  TGroupBoxEx = class (TCustomGroupBoxEx)
  published
    property Align;
    property Anchors;
    property BiDiMode;
    property Caption;
    property Color;
    property Constraints;
    property Ctl3D;
    property DockSite;
    property DragCursor;
    property DragKind;
    property DragMode;
    property Enabled;
    property Font;
    property ParentBiDiMode;
    property ParentColor;
    property ParentCtl3D;
    property ParentFont;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property TabOrder;
    property TabStop;
    property Visible;
    property OnClick;
    property OnContextPopup;
    property OnDblClick;
    property OnDragDrop;
    property OnDockDrop;
    property OnDockOver;
    property OnDragOver;
    property OnEndDock;
    property OnEndDrag;
    property OnEnter;
    property OnExit;
    property OnGetSiteInfo;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnStartDock;
    property OnStartDrag;
    property OnUnDock;
    property Advanced;
    property OnEnableDisableQuery;
    property OnPaintCaptionBackground;
  end;
procedure Register;
implementation
uses ComCtrls;
constructor TGroupBoxOptions.Create;
begin
  Inherited Create;
  fCaptionPos := gbTopLeft;
  fCaptionSpaces := True;
  fCaptionFont := TFont.Create;
  fCaptionFont.OnChange := CaptionFontChanged;
  fShowBorder := True;
end;
destructor TGroupBoxOptions.Destroy;
begin
  fCaptionFont.Destroy;
  Inherited Destroy;
end;
procedure TGroupBoxOptions.Changed;
begin
  if Assigned (fOnChange) then fOnChange (Self);
end;
procedure TGroupBoxOptions.CaptionFontChanged (Sender:
  TObject);
begin
  Changed;
end;
procedure TGroupBoxOptions.SetCaptionSpaces (Value:
   Boolean);
begin
  if fCaptionSpaces <> Value then begin
    fCaptionSpaces := Value;
    Changed;
  end;
end;
procedure TGroupBoxOptions.SetCaptionFont (Value: TFont);
begin
  fCaptionFont.Assign (Value);
  Changed;
end;
procedure TGroupBoxOptions.SetCaptionPos (Value:
  TGroupBoxCaptionPos);
begin
  if fCaptionPos <> Value then begin
    fCaptionPos := Value;
    Changed;
  end;
end;
procedure TGroupBoxOptions.SetShowBorder (Value: Boolean);
begin
  if fShowBorder <> Value then begin
    fShowBorder := Value;
    Changed;
  end;
end;
constructor TCustomGroupBoxEx.Create(AOwner: TComponent);
begin
  Inherited Create (AOwner);
  fOptions := TGroupBoxOptions.Create;
  fOptions.OnChange := OptionsChanged;
end;
destructor TCustomGroupBoxEx.Destroy;
begin
  fOptions.Free;
  Inherited Destroy;
end;
procedure TCustomGroupBoxEx.OptionsChanged (Sender:
TObject);
begin
  Invalidate;
end;
procedure TCustomGroupBoxEx.AdjustClientRect (var Rect:
TRect);
begin
  // Don't pass this on to Inherited
end;
procedure TCustomGroupBoxEx.CMEnabledChanged (var Msg:
  TMessage);
var
  Idx: Integer;
  Child: TControl;
{ CONTINUED ON FOLLOWING PAGE...}
```

```
{CONTINUED FROM PREVIOUS PAGE ...}
  Handled: Boolean;
begin
  Inherited;
  Invalidate;
  // Now enable or disable all the contained controls
  for Idx := 0 to ControlCount - 1 do begin
    Child := Controls [Idx];
    // Query application to see if we should do it
    Handled := False;
    if Assigned (fOnEnableDisableQuery) then
      fOnEnableDisableQuery (Self, Child, Enabled, Handled);
    if not Handled then begin
      Child.Enabled := Enabled;
      if Child.ClassName = 'TEdit' then begin
        if Enabled then
          TEdit (Child).Color := clWindow
        else
          TEdit (Child).Color := clBtnFace;
      end;
      if Child.ClassName = 'TListBox' then begin
        if Enabled then
          TListBox (Child).Color := clWindow
        else
          TListBox (Child).Color := clBtnFace;
      end;
      if Child.ClassName = 'TComboBox' then begin
        if Enabled then
          TComboBox (Child).Color := clWindow
        else
          TComboBox (Child).Color := clBtnFace;
      end;
      if Child.ClassName = 'TMemo' then begin
        if Enabled then
          TMemo (Child).Color := clWindow
        else
          TMemo (Child).Color := clBtnFace;
      end;
      if Child.ClassName = 'TDateTimePicker' then begin
        if Enabled then
          TDateTimePicker (Child).Color := clWindow
        else
          TDateTimePicker (Child).Color := clBtnFace;
      end;
      // Add your own type-specific preferences here?
    end;
  end;
end;

procedure TCustomGroupBoxEx.Paint;
var
  R: TRect;
  H2: Integer;
begin
  with Canvas do begin
    Font := fOptions.CaptionFont;
    H2 := TextHeight ('0') div 2 - 1;
    case fOptions.fCaptionPos of
      gbTopLeft..gbTopRight:
        R := Rect (0, H2, Width, Height);
      gbBottomLeft..gbBottomRight:
        R := Rect (0, 0, Width, Height - H2);
      gbLeftTop..gbLeftBottom:
        R := Rect (H2, 0, Width, Height);
      gbRightTop..gbRightBottom:
        R := Rect (0, 0, Width - H2, Height);
    end;
    if Ctl3D then begin
      Inc(R.Left);
      Inc(R.Top);
      Brush.Color := clBtnHighlight;
      if fOptions.ShowBorder then FrameRect(R);
      OffsetRect (R, -1, -1);
      Brush.Color := clBtnShadow;
    end else
      Brush.Color := clWindowFrame;
    if fOptions.ShowBorder then FrameRect(R);
    if Text <> '' then PaintCaption (Text);
  end;
end;

procedure TCustomGroupBoxEx.PaintCaption (Str: String);
var
  R: TRect;
  lf: TLogFont;
  tm: TTextMetric;
  BackgroundHandled: Boolean;
  X, Y, Flags, TH, TW: Integer;
begin
  if fOptions.CaptionSpaces then Str := ' ' + Str + ' ';
  if fOptions.CaptionPos in [gbLeftTop, gbRightBottom] then
    Str := Str + ' ';
  BackgroundHandled := False;
  TH := Canvas.TextHeight (Str);
  TW := Canvas.TextWidth (Str);
  // Deal with the easy stuff first !
  if fOptions.CaptionPos in [gbTopLeft..gbBottomRight] then
    begin
    R := Rect (8, 0, Width - 16, TH);
    if fOptions.CaptionPos in [gbBottomLeft..gbBottomRight]
      then OffsetRect (R, 0, Height - TH);
    Flags := dt_SingleLine;
    case fOptions.CaptionPos of
      gbTopLeft, gbBottomLeft: Flags := Flags or dt_Left;
      gbTopMiddle, gbBottomMiddle: Flags := Flags or
        dt_Center;
      gbTopRight, gbBottomRight: Flags := Flags or dt_Right;
    end;
    if Assigned (OnPaintCaptionBackground) then
      OnPaintCaptionBackground (Self, Canvas, R,
        BackgroundHandled);
    Canvas.Brush.Color := Color;
    if BackgroundHandled then SetBkMode (Canvas.Handle,
      Transparent);
    if Enabled then DrawText (Canvas.Handle, PChar (Str),
      -1, R, Flags) else begin
      SetTextColor(Canvas.Handle,
        ColorToRGB(clBtnHighlight));
      DrawText (Canvas.Handle, PChar (Str), -1, R, Flags);
      OffsetRect (R, -1, -1);
      SetBkMode (Canvas.Handle, Transparent);
      SetTextColor(Canvas.Handle,ColorToRGB(clBtnShadow));
      DrawText (Canvas.Handle, PChar (Str), -1, R, Flags);
    end;
  end else begin
    R := Rect (0, 8, TH, Height - 16);
    if fOptions.CaptionPos in [gbRightTop..gbRightBottom]
      then
      OffsetRect (R, Width - TH, 0);
    // This is only going to work with TrueType fonts....
    GetTextMetrics (Canvas.Handle, tm);
    if (tm.tmPitchAndFamily and tmpf_TrueType) = 0 then
      Exit;
    if Assigned (OnPaintCaptionBackground) then
      OnPaintCaptionBackground (Self, Canvas, R,
        BackgroundHandled);
    // Now build a new, vertical font.....
    GetObject (Canvas.Font.Handle, sizeOf (lf), @lf);
    if fOptions.CaptionPos in [gbLeftTop..gbLeftBottom] then
      lf.lfEscapement := 900
    else lf.lfEscapement := 2700;
    Canvas.Font.Handle := CreateFontIndirect (lf);
    Canvas.Brush.Color := Color;
    X := R.Left;  Y := R.Top;
    case fOptions.CaptionPos of
      gbLeftTop,    gbRightTop:     Y := 8 + TW;
      gbLeftMiddle, gbRightMiddle:  Y := ((Height - TW) div
        2) + TW;
      gbLeftBottom, gbRightBottom:  Y := Height - 16;
    end;
    if lf.lfEscapement = 2700 then begin
      Dec (Y, TW);  Inc (X, TH);
    end;
    if BackgroundHandled then SetBkMode (Canvas.Handle,
      Transparent);
    if Enabled then ExtTextOut (Canvas.Handle, X, Y, 0, Nil,
      PChar (Str), Length (Str), Nil) else begin
      SetTextColor (Canvas.Handle, ColorToRGB
        (clBtnHighlight));
      ExtTextOut (Canvas.Handle, X, Y, 0, Nil, PChar (Str),
        Length (Str), Nil);
      Dec (X);  Dec (Y);
      SetBkMode (Canvas.Handle, Transparent);
      SetTextColor (Canvas.Handle, ColorToRGB
        (clBtnShadow));
      ExtTextOut (Canvas.Handle, X, Y, 0, Nil, PChar (Str),
        Length (Str), Nil);
    end;
  end;
end;

procedure Register;
begin
  RegisterComponents('Experimental', [TGroupBoxEx]);
end;
end.
```

first calculated, assuming that the caption is in the `gbTopLeft` position. Next, the code tests to see if the caption string should be on the bottom of the group box and, if so, offsets the rectangle down to the bottom line. Finally, one of the `dt_Left`, `dt_Center` or `dt_Right` bit flags is `OR`'d into the `Flags` variable according to the required alignment of the caption.

This is another good reason for passing the entire side's-worth of bounding rectangle to the `OnPaintCaptionBackground` routine. By dealing with the entire potential length of the rectangle, we can offload the business of text positioning onto the Windows API routine by passing the alignment flags to the `DrawText` routine. This obviously wouldn't be possible if we were dealing with a tightly cropped bounding rectangle.

```
procedure PaintGradient(ACanvas: TCanvas; ARect: TRect;
  StartColor, EndColor: LongInt);
var
  Idx: Integer;
  sr, sg, sb, er, eg, eb: Byte;
begin
  sr := GetRValue (StartColor);
  sg := GetGValue (StartColor);
  sb := GetBValue (StartColor);
  er := GetRValue (EndColor);
  eg := GetGValue (EndColor);
  eb := GetBValue (EndColor);
  with ACanvas do for Idx := 0 to 31 do begin
    Brush.Color := RGB (sr + MulDiv (Idx, er - sr, 31),
      sg + MulDiv (Idx, eg - sg, 31), sb + MulDiv (Idx, eb - sb, 31));
    ACanvas.FillRect (Rect (ARect.Left, ARect.Top + MulDiv(Idx,
      ARect.Bottom - ARect.Top, 32), ARect.Right, ARect.Top +
      MulDiv(Idx + 1, ARect.Bottom - ARect.Top, 32)));
  end;
end;
procedure TForm1.GroupBoxEx1PaintCaptionBackground (Sender: TObject; Canvas:
 TCanvas; const Rect: TRect; var Handled: Boolean);
begin
  if TGroupBoxEx (Sender).Enabled then begin
    PaintGradient (Canvas, Rect, ColorToRGB (clWhite), ColorToRGB (clYellow));
    Handled := True;
  end;
end;
```

➤ *Listing 4*

```
procedure TForm1.RadioButton2Click(
  Sender: TObject);
begin
  GroupBoxEx2.Enabled :=
    Sender = RadioButton2;
  GroupBoxEx3.Enabled :=
    Sender = RadioButton1;
end;
```

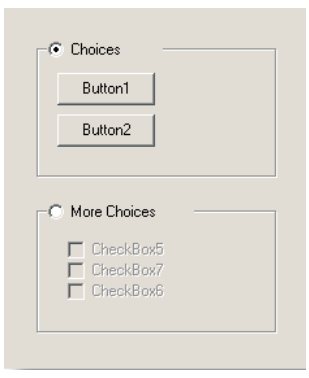
➤ *Listing 5*

The next job is to call the `OnPaintCaptionBackground` handler, if any. This renders the background image and sets the `BackgroundHandled` flag to indicate whether the application rendered the background. Why do we need this information? Look at it like this: if the application did provide a custom background, then we need to ensure that the caption is drawn *transparently*. If it isn't, then the caption will obliterate the background wherever text falls, which wouldn't look good. On the other hand, if the application didn't render a background, then we must ensure that the text is drawn *opaquely*. If it isn't, then the previously drawn group box border line will 'show through' the actual caption string and give an unpleasant strike-through effect.

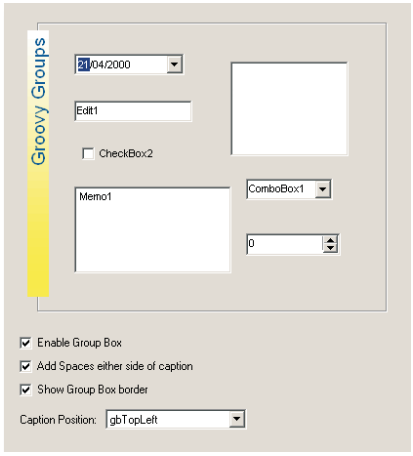
For this reason, the code checks to see if a background has been rendered and, if so, uses the `SetBkMode` API call to set transparent text drawing at the API level. It then makes a single call to `DrawText` for enabled group boxes (using the text colour specified in `CaptionFont`) and makes two calls to `DrawText` for disabled group boxes. This ensures that disabled captions are rendered using the familiar 'etched' look. Notice that, regardless of the value of the `BackgroundHandled` variable, the second call to `DrawText` must be done transparently for obvious reasons: we won't get the etched look if the second call obliterates the effect of the first!

Much the same sequence of events takes place with the vertical caption situation, except that we have to go to the extra trouble of creating a custom rotated font. As you'll appreciate, Windows can't rotate a bitmapped font (although now that everybody's using square pixels I can't honestly see why it doesn't) so the code simply exits without drawing the caption if a non-TrueType font is being used. Let me say that again: if you're not using a TrueType font, you won't see any caption when the text is oriented vertically.

As before, the application is given the opportunity of rendering a custom background, after which the code uses `CreateFontIndirect` to create the rotated font. The degree of rotation is 90 degrees (anticlockwise) for captions on the left side of the group box and 270 degrees for captions on the right side. This time round, because the text is being drawn vertically, we can't expect things like `DrawText` to handle the text alignment for us, so the vertical text case requires a little more effort.



➤ *Figure 4: The TGroupBoxEx component makes it easy to implement mutually exclusive sets of components such as the two group boxes here. In stealth mode, you don't even need to display the borders around each group box.*



➤ *Figure 3: An example of what can be done with a graduated fill behind the caption bar.*

Test-Bed Application

Figure 3 shows a souped-up group box as part of my little test-bed application. The group box contains a random assortment of controls, including a spin control. When the group box is disabled, all the controls are automatically greyed-out and the fancy graduated caption background disappears, to be replaced by an etched caption using the same font. For the sake of space, the full source code for the test-bed isn't included here, but you can see the important stuff in Listing 4. The `OnPaintCaptionBackground` checks to see if the group box is enabled and, if not, simply exits leaving the `Handled` flag set to `False`. If the box is enabled, it calls a simple

little `PaintGradient` utility routine to paint the white to yellow graduated stripe which you can see in the screenshot.

Because the event handler knows which group box it's working with (the `Sender` parameter), you could potentially give a different coloured stripe to each group box on a form. As ever, I'm not necessarily advocating the use of such pyrotechnics, I'm simply pointing out that the flexibility is there! Finally, you might question why I didn't build the graduated stripe effect into the group box itself. Again, it's a question of flexibility: some apps might want a simple solid colour, others might want to build a bitmap behind the caption; the choice is yours.

Figure 4 shows another aspect of the test-bed program. Some programmers like to use checkboxes or radio buttons to enable or disable group boxes. I toyed with the idea of building this facility into my group box control but didn't want to go overboard on the rampant featureitis front; maybe in version 2.0! Nevertheless, `TGroupBoxEx` makes it easier to implement this sort of user interface because of the ability to properly enable/disable child controls. With a shared handler for the two radio buttons in Figure 4, you don't really need to write any more code than that shown in Listing 5.

This can be extended to handle as many mutually-exclusive group boxes as you like.

That's it for this month. If you want to take this approach further, you might wish to implement a new version of `TRadioGroup` which inherits from `TCustomGroupBoxEx` rather than from `TCustomGroupBox`. Have fun!

Next month's *Beating the System* will be all about getting ready for Kylix, so don't miss it!

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. Contact Dave at TechEditor@itecuk.com