New Delphi 4 User Interface Features

by Warren Kovach

Delphi 4 adds a host of new features to make life easier for us programmers. Among these are some that simplify creating and maintaining a consistent and modern user interface. Do you fancy having your programs follow in the footsteps of Office 97 and the new Delphi 4 IDE by having new-style menus and draggable toolbars? Then read on...

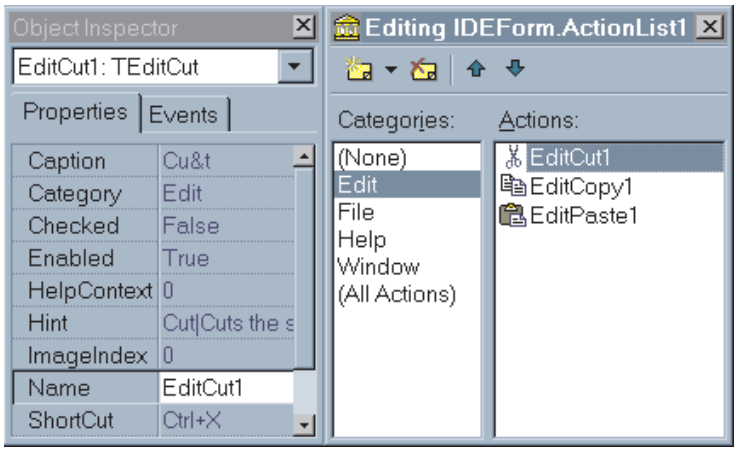
The most notable addition to the Delphi 4 IDE is the presence of bitmaps on the menus. Delphi 3 had no support for menu bitmaps, in fact it didn't even support owner draw menus, which would allow you to draw bitmaps yourself. The `TMenuItem` component in Delphi 4 adds support for the easy addition of bitmaps to menus.

There are no fewer than three ways to add bitmaps to your menus. The first is to load a bitmap for each menu item through its `Bitmap` property. This can get a bit tedious, however, so the component has a second method: the ability to be linked to a `TImageList`. Simply create an image list and add bitmaps to it, then set the `TMainMenu` (or `TPopupMenu`) `Image` property to point to the image list. Finally set the `ImageIndex` of each menu item to the appropriate number. Unfortunately, the bitmaps are not displayed in the menu designer, but they are visible in the actual menu on the form at design-time.

One of the key reasons for putting bitmaps on menus is that the same bitmap can be used on a toolbar button. Once the user has seen the bitmap next to a menu item, he or she will recognize the same bitmap on the corresponding toolbar button. For this to be of most use, however, the action performed by the menu item and button must be identical. This is where the third method of adding bitmaps comes in.

Lights, Camera, Action...

The new `TAction` and `TActionList` components are excellent tools for centralizing the behaviour and appearance of user interface elements. Each `TAction` corresponds to a single command, such as opening a file or copying data to the clipboard. It has an `OnExecute` event, which you would point to an event handler that actually performs the action. It also has several of the standard control properties, such as `Caption`, `Hint`, `Enabled` and `Visible`. You can link a set of actions (a `TActionList`) to a `TImageList`, which provides bitmaps to be associated with each action. Figure 1 shows the action list editor and the properties of one `TAction`.



➤ *Figure 1*

A number of the basic components, such as `TToolButton` and `TMenuItem`, have `Action` properties that can be linked to instances of `TAction`. So, if you have a file-saving `TAction` called `FileSaveAction` you could link it to both the `File | Save` menu item and the file-saving toolbar button. Both the menu item and the button then take on the caption, hint and bitmap set in the `TAction`. Clicking on either will call the same event handler.

If you need to disable certain actions you simply set the `Enabled` property of `FileSaveAction` to `False`: both the menu item and button will automatically follow suit. Likewise, any changes in the hint for `FileSaveAction` are reflected in the menu item and button. To make it easier to update the properties of a `TAction` an `OnUpdate` event is provided. Let's say you want to enable or disable `FileSaveAction` depending on whether the data has been modified. You create an `OnUpdate` event handler that checks the status of the data and sets `FileSaveAction.Enabled` appropriately. Every time your application goes into an idle state it will call all the `TAction.OnUpdate` methods to ensure the user interface is up to date.

This centralization of behaviour makes it easier to set up the standard user actions and modify them at runtime. Those of you who have complicated `UpdateMenuItems` methods that keep everything in sync will greatly appreciate these new components!

To create a new project using `TActionLists` you should first set up the `TImageList` and `TActionList` components, adding entries and bitmaps for all the appropriate actions. You can then rapidly populate your menus and toolbars by just adding new items and picking the appropriate action from the

Actions dropdown list on the Object Inspector. If you later decide to move some menu items to another menu this can be done rapidly by reassigning TActions to other entries: you don't need to copy or retype any properties.

Note that for the images to appear you must first make sure that the Images property of the main menu or the toolbar is set to the same TImageList as used by the action list. A side effect of a TImageList being linked to a menu is that Checked items will have a sunken 3D look rather than a simple checkmark.

Toolbars As Menus

Another new feature of the Delphi 4 IDE is the 3D menu. Each menu item now rises up as the mouse cursor passes over it and sinks down when it is pressed, just like the flat buttons on a TToolBar. After noticing this your first thought might be to look for a new property of TMainMenu called 3D, but you won't find it. The menu is not a standard Windows menu, but is emulated using a TToolBar.

Delphi 4 has added support to TToolBar and TToolButton to allow them to act like a regular menu. By using captions and not images on the buttons (and setting a few other properties, as explained below), we too can have 3D menus. They will behave like real menus, too, in that the usual keyboard navigation will be the same and the drop-down menus will track the mouse (that is, once you've clicked on one top-level menu the others will be pulled down automatically as the cursor passes over them).

Creating a toolbar-based menu is fairly simple, though slightly more fiddly than a regular menu. First, if your project does not already have a menu you should create and populate a TMainMenu component, just as you normally would. You can use links to a TActionList, as described above, to make this task easier. You must also set the Menu property of the form to be blank, so that this TMainMenu will not actually be displayed. Instead, it will provide the menu structure for the toolbar buttons.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i : integer;
begin
  Menu := nil;
  for i := pred(MainMenu1.Items.Count) downto 0 do
    with TToolButton.Create(ToolBar1) do begin
      MenuItem := MainMenu1.Items[i];
      Grouped := True;
      Parent := ToolBar1;
    end;
end;
```

➤ *Listing 1*

Next, add a TToolBar to your form, aligned to alTop. The Flat and ShowCaption properties must be set to True. You can then right click on the tool bar and choose New Button to add a series of buttons for each top-level menu item. For each button set its MenuItem property to the appropriate top-level menu item. For example, the first button will be the File menu, so you link it to File1, which is the default name for the first top-level menu item in most menus. The caption for that menu item will automatically be copied to the button.

You must set the Grouped property of each button to True. This allows the buttons to track the mouse cursor just like regular menus. You will also probably want to set AutoSize to True so that the buttons are just the right size for the caption. Run your project and you now have a 3D menu!

The process of linking the toolbar to a main menu can also be done at runtime by adding the code in Listing 1 to the form's OnCreate event handler. This is particularly useful for adding to a main form template, for use in creating new applications. Just add a blank TToolBar and TMainMenu to the template and add this code. Then when you create a new project from this template all you need to do is build the menu structure in the TMainMenu.

Finally, a warning. The menu-like appearance depends on having a recent version of Microsoft's COMCTL32.DLL: version 4.72 or later. This version is automatically installed on your machine when you install Delphi 4, but it may not be on your customer's machines. Also, if you install Microsoft's Internet Explorer 3 it will overwrite the DLL with an older version. When this happens the menu buttons will lose their flatness, will be the same width, and will display the & character in the caption. A setup program that installs the correct version is on the Delphi 4 CD-ROM in the directory \Info\Updates. You can also get it from Microsoft's website at

www. microsoft.com/msdn/downloads/files/40comupd.htm

Roll Your Own IDE

Let's try out the features we've looked at so far. Almost everything discussed in this article appears in the Delphi IDE, so the best test of the features is to create an application that emulates the IDE. The source code for the complete test application is on the disk in IDEClone.zip. To create this from scratch we simply create a new application with File | New Application, then add a TMainMenu, TActionList and TImageList, linking them together and populating them with actions and bitmaps as described above.

We will next add a bar across the top for the new style menu as well as other toolbars. You may have noticed that the sliding, moveable toolbars in the Delphi IDE have a somewhat different appearance and behaviour than the usual TCoolBar from the Win32 palette. This is because it is created using the new TControlBarcomponent, which can be found on the Additional palette page. This has some important enhancements over the TCoolBar and, unlike TCoolBar, is not based on the Microsoft common control in COMCTL32.DLL. We will add one of these to the form, making sure that Align is set to alTop and AutoSize to True.

Then we can add a TToolBar to the TControlBar, aligned to alTop, to act as the menu; we will add buttons and link them to the main

menu as described above. This main form should be shown as a bar across the top of the screen, like the Delphi IDE. Also, we must make sure the form is always just large enough to contain the control bar. To do this we create an `OnShow` event handler like that in Listing 2.

Note the setting of the `Constraints` property for the form. This is a new property in Delphi 4 that makes it much easier to set maximum and minimum sizes of controls and forms. Previously you had to trap the `WM_GETMINMAXINFO` message to constrain sizes. With this set the user cannot change the height of the form. However, the `TControlBar` might need to resize itself when the various toolbars are moved around. If it does this, the form must be resized too. To allow this we must add an `OnResize` event handler to the `TControlBar` and insert the code in Listing 3.

This removes the constraint, resets the form's height to accommodate the new control bar size, then resets the constraint. Now compile and run this project and you will see a nice 3D menu at the top of the screen.

Floating Toolbars

Let's add two other toolbars to contain some buttons, one for some File menu actions and one with Edit commands. We will first need to set the `AutoSize` property of the control bar to `False` so that we can resize it to make room for the toolbars. Drop a couple of toolbars on the control bar and set their properties as follows: `Flat` is `True`, `DragKind` is `dkDock`, `DragMode` is `dmAutomatic` and `Images` points to the form's `TImageList`. You can now add buttons by right clicking on the toolbar, choosing `New Button`, and setting the buttons `Action` property to the appropriate action. Now set the control bar's `AutoSize` and `DockSite` properties to `True`.

When you run the application you will see the two new toolbars. When you drag them by their handles you can rearrange the toolbars on the control bar. If you drag them off the control bar they will automatically turn into floating toolbars, complete with enclosing

```
procedure TIDEForm.FormShow(Sender: TObject);
begin
  Left := 0;
  Top := 0;
  Width := Screen.Width;
  Height := ControlBar1.Height + (Height - ClientHeight);
  Constraints.MaxHeight := Height;
end;
```

➤ *Listing 2*

```
procedure TIDEForm.ControlBar1Resize(Sender: TObject);
begin
  Constraints.MaxHeight := 0;
  Height := ControlBar1.Height + (Height - ClientHeight);
  Constraints.MaxHeight := Height;
end;
```

➤ *Listing 3*

window, a caption bar and a close button.

This behaviour is enabled by the setting of the `DragKind` property to `dkDock`. Docking is a new feature of Delphi 4 that is pervasive throughout the VCL. It allows any control that is set to `dkDock` to become, at runtime, a child control of another that is designated a dock site. This allows the user to rearrange controls at will. Delphi 4 has a built-in docking manager that controls much of this behaviour automatically. In the case of our toolbars they are being dragged off their parent control but they are not being dropped onto another designated dock site. When this happens the docking manager creates a new form to act as a parent to the toolbar.

Note that we've set the `DockSite` property of the control bar to `True`. This means that the control bar can host any dockable control that is dropped on it. To test this just drag one of the floating tool bars back up to the control bar. It will be docked back onto the control bar again. The floating toolbar window, which is not needed any more, will disappear.

Dockable Tool Windows

In the Delphi IDE almost everything is dockable. The toolbars can be dragged out to form floating toolbars, as is done in our example project. Also, most of the other tool windows, such as the object inspector or project manager, can be docked to each other. In fact, in my opinion the tool windows are just too ready to jump into bed with each other at the slightest flick of the mouse. At first I found it a bit disconcerting, although it's growing on me. Fortunately it can be turned off by right clicking on a window and choosing the `Dockable` menu item.

There are a number of ways to create docking windows such as these, ranging from the virtually automatic to schemes where the programmer has complete control over where and how docking occurs. The simplest method is to create a dock site using a `TPanel`.

Let's say we want to create something that works in a similar way to the code editing window in the IDE. This always has a series of `TRichEdit` editing controls on a tabbed control. It also allows other windows to be docked at either side or at the bottom. The size of the editing controls will be adjusted so that they and the new docked control can both be accommodated within the same window.

To demonstrate this we will add three new forms to our IDEClone project. Two of them will be tool windows. For this demonstration they can be blank forms, although in the project on the disk I've made them up to look a bit like the Object Inspector and Project Manager of the IDE. The important step, though, is that the two forms must have `DragKind` set to `dkDock` and `DragMode` set to `dmAutomatic`.

Now create a third form that will house the editing controls. First we need to create a dock site. Add

a TPanel and set its Align property to alRight and DockSite to True, so that it can accept dockable controls. We only want this panel to be visible when something is actually docked on it, so we will set AutoSize to True. Now add another TPanel with Align set to alClient to fill the rest of the window. To this you can add a TRichEdit (or perhaps a TPageControl with several editors).

We want all three windows visible at program startup so, in the FormShow method of the main form, we will add calls to the Show method of each window. Now when we run the program we will see the three tool windows as well as the menu bar at the top. Drag one of the dockable forms over to the right side of the edit window. When the cursor comes within a few pixels of the right side (within the 'zone of influence' of the autosizing TPanel, which now has a width of 0) the dragging rectangle will change shape and position to show where the form will be docked if it is dropped. Drop it and it will be docked into the editing window.

If you drag and drop the second form onto the right side of the editing window you will see the docking area being divided between the two forms, either horizontally or vertically. The built-in docking automatically does this and places a splitter bar between the two to allow you to adjust the space allocated to the two windows.

Taking Control

Unfortunately, this automatic docking with a TPanel does not allow you to adjust the relative amount of space allocated to the docking area and the editor. No splitter is added automatically between the panel and the rich edit control. You might try adding one at design-time. However, this will not work reliably because the panel is resized at runtime to accommodate the docked control. When this happens the splitter remains at the right edge of the form, not between the two areas. Instead we must adjust the positioning of the splitter in code at runtime.

Another problem is that the docked control will always be the same height or width as it was when it was a separate window. This means that it may take up too much of the editing window when docked. We will initially want to limit it to just one third of the window size; the user can adjust this later.

The docking support adds a number of new events that allow you to take more control of the docking process. There are two, OnDockOver and OnDockDrop, that will help us add the splitter, as well as customize the action a bit more. To test these we will add two new panels, one aligned to the left and one aligned to the bottom (you must first set the Align property of the panel containing the rich edit control to alNone and resize it to give a bit of room for working). We will set their DockSite properties to True but leave the AutoSize properties set to False. We will also set the width of the one at the left and the height of the bottom one to 0. Next add two TSplitters, one aligned to the left and one to the bottom, and set their Visible attributes to False.

We next add the event handlers. OnDockOver will be fired when a dockable control is over a dock site. We can use this to adjust the docking rectangle shown when the form is over the dock site. The LeftDockPanelDockOver and BottomDockPanelDockOver methods in Listing 4 show how to do this for panels at the bottom and left.

OnDockDrop is called when the control is actually dropped onto the dock site. It is here that we do the adjustments to the panel and splitter. The first statement of the DockPanelDockDrop method checks if this is the first control to be dropped on this panel; the docking manager will take care of adding splitters between two or more controls. If it is first then the ShowDockPanel method is called. This is used for hiding the panel

➤ *Listing 4*

```
procedure TEditWin.ShowDockPanel(
  APanel: TPanel; MakeVisible: Boolean);
begin
  if APanel = LeftDockPanel then
    LeftSplitter.Visible := MakeVisible
  else
    BottomSplitter.Visible := MakeVisible;
  if MakeVisible then
    if APanel = LeftDockPanel then begin
      APanel.Width := ClientWidth div 3;
      LeftSplitter.Left := APanel.Width+LeftSplitter.Width;
    end else begin
      APanel.Height := ClientHeight div 3;
      BottomSplitter.Top := ClientHeight - APanel.Height -
        BottomSplitter.Width;
    end
  else
    if APanel = LeftDockPanel then
      APanel.Width := 0
    else
      APanel.Height := 0;
end;
procedure TEditWin.DockPanelDockDrop(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer);
begin
  if (Sender as TPanel).DockClientCount = 1 then
    ShowDockPanel(Sender as TPanel, True);
  (Sender as TPanel).DockManager.ResetBounds(True);
end;
procedure TEditWin.DockPanelUnDock(Sender: TObject; Client:
  TControl; NewTarget: TWinControl; var Allow: Boolean);
begin
  if (Sender as TPanel).DockClientCount = 1 then
    ShowDockPanel(Sender as TPanel, False);
end;
procedure TEditWin.LeftDockPanelDockOver(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean);
var ARect : TRect;
begin
  ARect.TopLeft := LeftDockPanel.ClientToScreen(Point(0,0));
  ARect.BottomRight := LeftDockPanel.ClientToScreen(
    Point(Self.ClientWidth div 3, LeftDockPanel.Height));
  Source.DockRect := ARect;
end;
procedure TEditWin.BottomDockPanelDockOver(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean);
var ARect : TRect;
begin
  ARect.TopLeft := BottomDockPanel.ClientToScreen(
    Point(0, - Self.ClientHeight div 3));
  ARect.BottomRight := BottomDockPanel.ClientToScreen(
    Point(BottomDockPanel.Width, BottomDockPanel.Height));
  Source.DockRect := ARect;
end;
procedure TEditWin.DockPanelGetSiteInfo(Sender: TObject;
  DockClient: TControl; var InfluenceRect: TRect;
  MousePos: TPoint; var CanDock: Boolean);
begin
  CanDock := DockClient is TForm;
end;
```

and splitter when the control is undocked as well as showing them when they are docked, so we pass a boolean `MakeVisible` parameter along with the dock site object. Within the method we first check which panel is receiving the control and make the appropriate splitter visible. Next, if we are docking, we resize the panel so that it takes up one third of the editing window and position the splitter between the panel and the editor. After the call to `ShowDockPanel` we force a repaint of the docked control by calling the `ResetBounds` method of the dock site's docking manager.

Undocking of the control is handled through the `DockPanelUndock` handler for the `OnUndock` event. Here we simply check to see if this is the last control being undocked from the panel, then call `ShowDockPanel` to hide the panel and splitter.

You may wish to only dock certain types of docking controls onto a site. For example, the IDE will not allow toolbars to be docked onto other windows: they remain floating toolbars no matter where they are. We can do the same using the `OnGetSiteInfo` event. This is fired just before `OnDockOver`. Within a handler for this event (`DockPanelGetSiteInfo` in Listing 4) we can check the `DockClient` parameter to see if the dockable control is one we want (a `TForm` descendant in this case). If it is we set the `CanDock` parameter to `True` and docking will continue. If not we set it to `False` and the panel will act as if it is not a dock site; no docking preview rectangle will be drawn and `OnDockOver` will not be invoked.

Combining Tool Windows

What we've done so far emulates what happens in the Delphi IDE when you drag a tool window onto the main editing window. However, you can also combine two or more tool windows into a single window, either as tiled windows or as separate pages on a tabbed control. Doing this in our own programs is more difficult. We must take over control of docking at an even lower level, pre-empting the default docking manager. We can then examine the position of the window to determine which type of docking is required (tiled or tabbed), create a new form of the required type to accommodate both tool windows, then dock them together.

Delphi 4 comes with a demo project (Demos\Docking\dockex.dpr) that shows how to do this kind of docking. To add this functionality to our own program we can just copy three of the forms in that project (the .PAS and .DFM files for `DockForm`, `TabHost` and `ConjoinHost`) to our own project and make three changes to Dockform.PAS. First, copy the `VisibleDockClientCount` function from the docking example's MAIN.PAS file to this unit. Second, search for `ShowDockPanel` and change the qualifying form to be our editing window `EditWin`. Lastly, change `Main` in the second `Uses` clause to `Editor`, the name of the editing window's unit. Next we modify our tool windows' declarations so they are descended from `TDockingForm` rather than `TForm`. Run the project and you will now be able to dock the two tool windows together, just as in the IDE.

Let's look at how this docking works. The key to it is that we capture the `CM_DockClient` message sent by Delphi when docking is about to occur. This is normally captured by the `TWinControl.CMDockClient` method but we will replace this with our own method. Within this method we check the position of the mouse, using the `TDockableForm.ComputeDockingRect` method, to determine whether to dock the windows tiled or as tabbed pages.

If the mouse is in the middle of the form then we assume the user wants to dock the forms as tabbed pages. We then create an instance of `TTabDockHost`. This is a simple form with a `TPageControl` set up as a dock site. The control's `DockSite` property is `True` and it has `OnGetSiteInfo` and `OnDockOver` event handlers that ensure only `TDockableForms` can be docked. Once this is created we dock both the tool windows to this page control by calling the `ManualDock` method for each.

If the mouse is closer to one of the form's sides then we will tile the windows. In this case we create an instance of `TConjoinHost`, a blank form with no controls and with its own `DockSite` set to `True`. We will dock one of the tool windows to this, then dock the second one to the appropriate side by passing the `DockType` value (a `TAlign` returned by `ComputeDockingRect`) to `ManualDock`.

If this isn't enough control over the docking process for you then you can go to an even lower level and completely replace the default docking manager with one of your own design. You can then use the `DockManager` property of your controls and forms to point to your own docking manager. This approach is not for the faint hearted, though. Have a look in the VCL source code module CONTROLS.PAS and study the `TDockTree` and `TDockZone` classes to see what is involved.

Other New Goodies

There are several other new components and properties that enhance the user interface design capabilities of Delphi. Two new Windows common controls are encapsulated in the VCL. `TMonthCalendar` displays a monthly calendar and allows the user to select a date or a range of dates. `TPageScroller` lets you place a large control into a small space. It can contain another `TWinControl` descendant. If the space provided by the page scroller is too small to display the whole control then scroll buttons appear allowing you to scroll back and forth (or up and down) along the control. Windows 98 and Internet Explorer 4 use this common control to fit long menus into a limited space.

`TPageControls` and `TTabControls` can now have bitmaps on their tabs, implemented through the `Images` properties link to a `TImageList`. They also have a `RaggedRight` property that lets you determine, when there is more than one row of tabs, whether they should be spaced out along the top of the entire control or remain at their normal widths. A new `Style`

property lets you have the tabs displayed as flat or raised buttons instead.

`TTrackBar` adds more customization features through the `SliderVisible` and `ThumbWidth` properties. `TProgressBar` adds `Smooth`, to produce a solid bar rather than a segmented one, and an `Orientation` property. `TListView` now has a `FlatScrollBars` property and `TTreeView` a `AutoExpand` property.

Finally, a few new properties occur throughout the VCL hierarchy. All `TWinControl` descendants now recognise the mouse wheel messages from Windows and fires events that the programmer can trap and process. They all also have `BevelEdge`, `BevelWidth` and `BorderWidth` properties to customize the edges and borders of the controls. Support is also added for Middle Eastern languages that are read from right to left. The `BiDiMode` property enables typing of text from right to left when the appropriate input locales are used. The `FlipChildren` method and `Edit | Flip Children` IDE menu option swaps the positioning of controls around the vertical axis so that they can be followed by the user in a right to left fashion, matching the flow of reading.

Warren Kovach has been an Action Man since Delphi 4 was released. He also writes and sells statistical software and is the author of *Delphi 3 – User Interface Design*, published by Prentice Hall. You can email him at wlk@kovcomp.co.uk or visit www.kovcomp.co.uk