Creating Windows Vista Ready Applications with Delphi
Update - January 2007

Reintroduction

I was overwhelmed, to say the least, by the reception my original article on creating Windows Vista ready applications with Delphi received. I was very flattered to be included on the Delphi Hour (replay available here).

This article aims to provide updated and additional information on creating Vista ready applications with Delphi. There are a few small issues with the code presented in the original article which we will examine and fix, some new code for working with fonts we will explore, code that allows us to take advantage of the new TreeView style in Vista, and more.

This article will assume you've read the first article, and the code samples pick up where the last ones left off.

Font Fixes

Thanks goes to David Rose for helping find a problem with SetVistaFonts(). I was previously under the impression that setting Font.Name to a font that wasn't installed would result in the Font.Name not being changed. This is not the case. Our original code checked Font.Name to see if Segoe UI was successfully applied to the form font, which doesn't work. Our new code checks Screen.Fonts to see if Segoe UI is an available font.

```
procedure SetVistaFonts(const AForm: TCustomForm);
begin
  if (IsWindowsVista or not CheckOSVerForFonts)
    and not SameText(AForm.Font.Name, VistaFont)
    and (Screen.Fonts.IndexOf(VistaFont) >= 0) then
  begin
    AForm.Font.Size := AForm.Font.Size + 1;
    AForm.Font.Name := VistaFont;
  end;
end;
```

The new code also introduces the global variable CheckOSVerForFonts, which determines if the font changes should only take affect under Vista or if they should work under any OS. The original code wouldn't have exhibited any issues unless Segoe UI was not an installed font under Vista (not likely), but this code is much more complete and correct.

Font Additions

On the CodeGear Developer Network, Daniel England brought up the fact that he didn't like the font names being hard coded. While I agree, my intent was to try to keep most of these changes in-line with the out-of-box experience found with Delphi. Delphi, by default, does not adjust form fonts to match the current Windows fonts. If it did, we wouldn't need to change our fonts to Segoe UI under Vista at all.

That said, the first addition to the font code is SetDesktopIconFonts(). This can be called, passing in a TFont, and the font object will have its properties adjusted to match the current desktop icon font setting in Windows. In the updated sample, this method is used rather than SetVistaFonts().

```
procedure SetDefaultFonts(const AFont: TFont);
begin
  AFont.Handle := GetStockObject(DEFAULT_GUI_FONT);
end;

procedure SetDesktopIconFonts(const AFont: TFont);
var
  LogFont: TLogFont;
begin
  if SystemParametersInfo(SPI_GETICONTITLELOGFONT, SizeOf(LogFont),
    @LogFont, 0) then
    AFont.Handle := CreateFontIndirect(LogFont)
  else
    SetDefaultFonts(AFont);
end;
```

This code tries to get the font settings for the current desktop icon font in Windows and, failing that, falls back on the default font settings for Windows.

Our second addition to the font code is SetVistaContentFonts(). This does what SetVistaFonts() does, only for content, such as memos, rich edits, and the like. Here, the new de facto font for the Vista application is Calibri.

```
procedure SetVistaContentFonts(const AFont: TFont);
begin
  if (IsWindowsVista or not CheckOSVerForFonts)
    and not SameText(AFont.Name, VistaContentFont)
    and (Screen.Fonts.IndexOf(VistaContentFont) >= 0) then
  begin
    AFont.Size := AFont.Size + 2;
    AFont.Name := VistaContentFont;
  end;
end;
```

Calling this new method and passing in a TFont object will result in the font name being changed to Calibri and the font size being increased if the application is running under Vista (or the CheckOSVerForFonts global variable is false) and the Calibri font is installed.

Vista Style TreeView

Another of the UI elements updated in Windows Vista, and not mentioned in the original article, is the TreeView. Under Windows Vista, TreeView selections are a light blue gradient rather than the traditional selection rectangle, and the TreeView buttons are small triangles rather than pluses.

Getting our TTreeView objects to use this new style is fairly trivial, and the code is found in our updated sample in the method SetVistaTreeView().

```
procedure SetVistaTreeView(const AHandle: THandle);
begin
  if IsWindowsVista then
    SetWindowTheme(AHandle, 'explorer', nil);
end;
```

Calling this new method, and passing in a TreeView's Handle, will result in the TreeView being rendered with the new Vista style.

PopupParent Workaround

Our original article introduced code to work around Delphi's hidden application window, allowing several of Vista's new UI elements to function properly, such as Flip 3D and live thumbnails. However, these changes left us with the need to set the PopupParent property proceeding each call to ShowModal to ensure proper Z ordering of our forms.

Correspondence with Eric Fortier probed me to dig a bit further and see if it was possible to eliminate this requirement. While this is indeed possible, it requires changes to VCL code (similar to our changes to Dialogs.pas in the original article):

- First, make a copy of Forms.pas and put that copy either in our application's source directory or somewhere on our library path.
- Second, in that copy of Forms.pas, go to line 4032
- Look for a case statement involving LPopupMode
- Comment out the first line following the initial case: *//pmNone: Application.Handle;*
- Edit the second line, which currently just catches pmAuto, to catch pmNone as well: *pmNone, pmAuto:*
- Save your changes to your copy of Forms.pas

This should leave you with a case statement around line 4032 that checks LPopupMode and now takes the same action for both pmNone and pmAuto rather than using Application.Handle as the parent when PopupMode is pmNone. This is just what we want. Now, the VCL will try to find our parent form by detecting the active window with a bit of logic, rather than just using Application.Handle (which won't work due to our changes from the first article) when PopupMode is pmNone (the default).

Strange Activation

One side affect to the code introduced in the original article for fixing window animations and previews is that a form beneath another modal form can become visually active if its taskbar button is clicked. Note that this is just a visual discrepancy; you cannot actually interact with anything on the covered form. Max Pyatnitsky brought up this problem on the CodeGear newsgroups, and I thought I'd share a slightly modified version of his solution (this is also now included in the updated sample and compiled executable):

```
procedure TMainForm.WMActivate(var Message: TWMActivate);
begin
  if (Message.Active = WA_ACTIVE) and not IsWindowEnabled(Handle) then
  begin
    SetActiveWindow(Application.Handle);
    Message.Result := 0;
  end else
    inherited;
end;
```

Thanks to Max Pyatnitsky for his solution to this problem. This passes the activation on to the hidden application form when our form is activated, ensuring any modal forms get focus properly.

Enumerating Windows

There is another caveat that may affect users who have implemented the code to work around Delphi's hidden application window. According to Tom Nagy on the CodeGear newsgroups, developers that enumerate top level windows using EnumWindows() may be using code such as the following as the EnumWindowsProc callback:

```
if IsWindowVisible(Wnd) and
  ((GetWindowLong(Wnd, GWL_HWNDPARENT) = 0) or
  (HWND(GetWindowLong(Wnd, GWL_HWNDPARENT)) = GetDesktopWindow)) and
  ((GetWindowLong(Wnd, GWL_EXSTYLE) and WS_EX_TOOLWINDOW) = 0) then
begin
  ...
end;
```

According to Tom, this may be a fairly common Delphi code-snippet used when enumerating windows. However, our change to our forms' CreateParams flags will cause the check for WS_EX_TOOLWINDOW to fail. Tom proposes the following code change (removing the check for WS_EX_TOOLWINDOW) to fix the problem:

```
if IsWindowVisible(Wnd) and
  ((GetWindowLong(Wnd, GWL_HWNDPARENT) = 0) or
  (HWND(GetWindowLong(Wnd, GWL_HWNDPARENT)) = GetDesktopWindow)) then
begin
  ...
end;
```

Thanks to Tom Nagy for catching this and reporting it on the newsgroups. Hopefully this will help others avoid what could be a hard-to-catch bug!

ALT Shortcuts Hiding Controls

There is another bug that strikes Delphi applications when they are run under Vista. This bug has been reported on Quality Central and has several workarounds available there. The problem is that, when the ALT key is pressed, some TWinControl descendants may disappear from your form until they are forced to repaint. There is a component available here which fixes this problem. I tried the code fixes for Controls.pas presented here and they did not solve the problem for me. The above component seems to work well though.

A Mutli-Platform Task Dialog

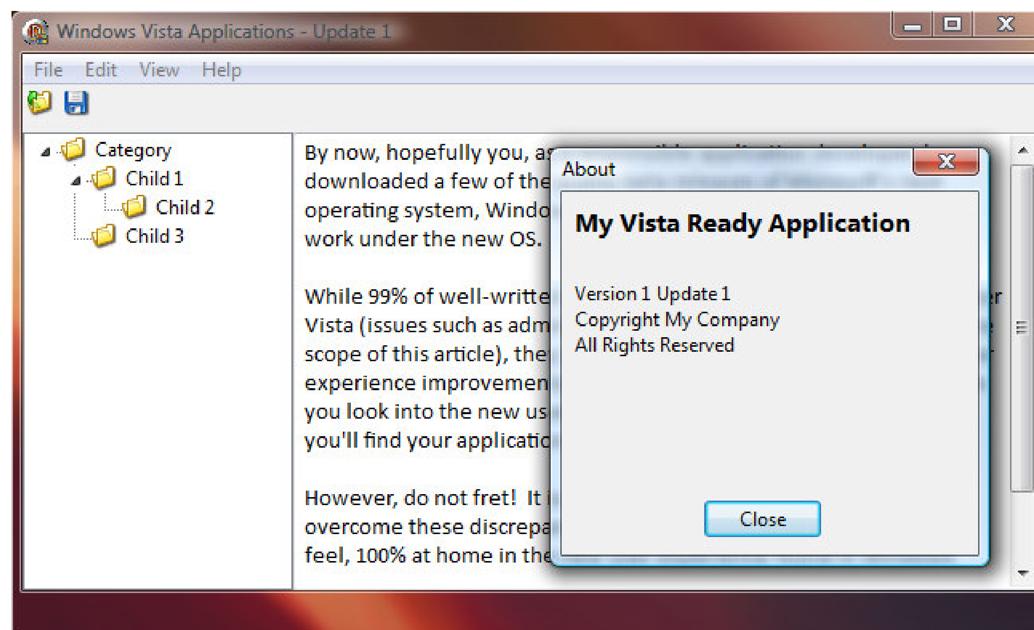
The original article introduced Vista's concept of Task Dialogs, and how to use these new style dialogs from our Delphi applications. While these new Task Dialogs are a great addition to Windows Vista, it isn't easy to support both the new Task Dialog features and fall back on some other supported dialog under XP and earlier.

Thankfully, the guys at TMS have done the work for us! Building on the great work they did working with Vista's Task Dialogs, they have released a component called TTaskDialog. I don't want this article to come off as a veiled advertisement, but since our original article specifically targeted the Task Dialog (among other UI elements), I felt this component was worth a mention.

The TTaskDialog component takes the work TMS had already done in exposing all of the advanced functionality found in the Task Dialog API, and builds in hard-coded support for legacy OS's such as XP and earlier. This means developers can take advantage of the Task Dialog today and keep supporting earlier versions of Windows.

Conclusion & Outstanding Issues

All these changes, taken together, give us an application that improves on the Vista support discussed in the original article. Our fonts will behave as they should have when Segoe UI is not available, and can optionally change to fit the Windows desktop icon font. We can now have our content areas support a newer font style and our TreeView controls styled in accordance with the new Vista UI elements.



Download Updated Sample Code
Download Final Compiled Executable

However, we are left with one outstanding issue, and any feedback or potential solutions for this are very welcome (contact information is in the page footer). If you combine the code found in the original article to support the minimizing and maximizing animations with the code found in this article for supporting the new TreeView UI elements, your application will end up with a small, but noticeable, bug. Minimizing a form will result in that form first disappearing behind any window immediately behind it before proceeding to animate to the taskbar. This bug can also be seen in some Microsoft applications, so it is hard to tell what may be causing it. Any feedback is welcome and credit will be given for any solutions.

Below find an expanded list of links to articles and tutorials involving Windows Vista and user interfaces. Thanks again to everyone for their feedback and kind reception of the original article. I hope this update will prove a valuable addition.

What's New in Windows Vista
Top Rules for the Windows Vista User Experience
Top Guidelines Violations
Top 10 Ways to Light Up Your Windows Vista Apps
The new File Open / Save Dialogs in Windows Vista
Using the new Windows Vista Task Dialog
Taking the New Windows Vista Task Dialog One Step Further
Full Featured Multi-Platform Task Dialog Control

Copyright © 2007 Victory Technologies, Inc.
Questions? Comments? Contact Nathanial Woolls