Make Your Menus Pop

Doug Hennig

Last issue, Doug discussed ctl32_ContextMenu, an object-oriented menu class that's part of the ctl32 library. This month, he looks at another OOP menu class, this time the VFPX PopMenu project.

I haven't used VFP's native menus directly in many years. I always hated the fact that the Menu Designer is a clunky tool and that menus are hard-coded procedural code rather than object oriented. Because I wanted a more flexible menuing system, I created a set of OOP menu classes that are now part of VFPX (http://tinyurl.com/2a2jnak). Internally, they're just wrappers for the VFP DEFINE MENU, DEFINE PAD, DEFINE BAR, and other menu-related commands, but at least I can manipulate my menus as objects now.

However, the other issue is that, as you can see in **Figure 1**, VFP's native menus (the one on the left) look out of date compared to menus in more recent applications, such as Microsoft Outlook (the one on the right).

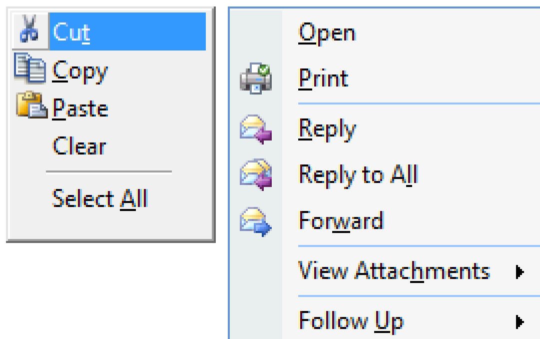

Figure 1. VFP's native menus look out of date compared to newer applications.

Fortunately, a VFPX project called PopMenu, written by LingFeng Shi, provides an entirely new way of doing menus. Not only are they object-oriented, they also use the native Windows menuing system rather than VFP's menuing system, meaning that your application's menus can look just like those in Microsoft Office applications.

One downside of PopMenu is that there's no documentation and all comments (in code and the Properties window) are in Chinese. So, I dug through the code, played with the samples, and found that I really like this class.

To use PopMenu, download it from VFPX (http://tinyurl.com/2c9ecr2), unzip it into any folder, and add the main class library, VCX_Tools.VCX, to your project.

Note that PopMenu is primarily for shortcut rather than system menus. I'll discuss this more in detail later.

Creating a menu

To create a shortcut menu, instantiate PopMenu or a subclass, then add items to it using either Add or AddItem.

AddItem is the simpler of the two methods: AddItem(cTitle, cCommand, vEnabled, nFlag). cTitle is the caption for the item, cCommand is the command to execute when the item is selected, vEnabled is an expression which indicates whether the item is enabled or not, and nFlag is a numeric value indicating how the item should appear (discussed later). These parameters, all of which are optional, match up with properties of menu item objects discussed later. Here's an example:

```
loMenu = newobject('PopMenu', 'VCX_Tool.vcx')
loMenu.AddItem('\<New')
loMenu.AddItem('\<Open')
loMenu.AddItem('\<Save')
```

Add provides more flexibility: Add(cParentKey, cKey, cTitle, cCommand, cPicture, vEnabled, nFlag). cParentKey is the key assigned to the parent item for this item and cKey is the key assigned to this item; these parameters are used when you want to create a submenu. cTitle, cCommand, vEnabled, and nFlag have the same meaning as for AddItem. cPicture is the name of an image file to use for the picture for the item. All parameters are optional. Here's an example:

```
loMenu = newobject('PopMenu', 'VCX_Tool.vcx')
loMenu.Add('', '', '\<New',  ;
    'messagebox("You chose New")', ;
    'Images\New.bmp')
loMenu.Add('', '', '\<Open', '', ;
    'Images\Open.bmp')
loMenu.Add('', '', '\<Save', '', ;
    'Images\Save.bmp')
```

Both Add and AddItem create a _MenuInfo object, set its properties, add it to an internal

collection, and return a reference to the object. This object has the following properties you can set as necessary:

- cCommand: the command to execute when the item is selected. This isn't really needed because you can use a CASE statement after displaying the menu to decide what to do.
- cEnabled: an expression which indicates whether the item is enabled or not. This can either be a logical value or an expression which evaluates to a logical value. For example, "GetUserRights('Payroll')" calls the GetUserRights function to determine whether the user has rights to payroll and disables the item if not.
- cKey: the key for the item. If you don't assign a key, a SYS(2015) value is used.
- cMessage: the status bar text for the item.
- cParentKey: the key for the parent item for this item. If this contains a valid key, this item appears in a submenu of the parent item.
- cPicture: the name of an image file to use for the item.
- cTitle: the caption for the item. Use "\<" or "&" to indicate that the following character is a hotkey. Use "\-" for a separator line.
- nFlags: there are several constants defined in WIN32API.H you can use for nFlags:
    - MF_MENUBARBREAK (0x20): places the item in a new column without a vertical separator between columns.
    - MF_MENUBREAK (0x40): like MF_MENUBARBREAK but separates the columns with a vertical line.
    - MF_CHECKED (0x8): displays a checkmark next to the item. This is ignored if "owner drawn" menus are used (discussed later).
- nHeight: the height of the bar in pixels. The default is the same as the nItemHeight property of PopMenu, which I'll discuss later.
- nIndex: the order for the item (1 for the first item, 2 for the second, etc.).
- nWidth: the width of the bar in pixels. The default is the same as the nItemWidth property of PopMenu, which I'll discuss later.
- oRefMenu: if this contains a reference to another PopMenu object, that object is used as a submenu for this item. If you set this, be sure to call CreateContext(oItem, oItem.oRefMenu) to properly initialize the submenu.

PopMenu contains a few helper methods:

- SetMessage(cMessage [, nOrder]): sets the message for the specified menu item; if nOrder isn't specified, the last added item is used.
- SetPicture(cImageFile [, nOrder]): sets the picture for the specified menu item; if nOrder isn't specified, the last added item is used.
- Clear(): removes all items from the menu.

To display the menu, call Show. Here's an example that displays a typical "edit" menu. Each item is enabled if the appropriate function in the Edit menu is enabled and performs the same action when selected.

```
loMenu = newobject('PopMenu', 'VCX_Tool.vcx')
with loMenu
    .Add('', '', 'Cu\<t', ;
        "sys(1500, '_MED_CUT', '_MEDIT')", ;
        'Images\CutXPSmall.bmp', ;
        "not skpbar('_MEDIT', _MED_CUT)")
    .Add('', '', '\<Copy', ;
        "sys(1500, '_MED_COPY', '_MEDIT')", ;
        'Images\CopyXPSmall.bmp', ;
        "not skpbar('_MEDIT', _MED_COPY)")
    .Add('', '', '\<Paste', ;
        "sys(1500, '_MED_PASTE', '_MEDIT')", ;
        'Images\PasteXPSmall.bmp', ;
        "not skpbar('_MEDIT', _MED_PASTE)")
    .Add('', '', 'Clear', ;
        "sys(1500, '_MED_CLEAR', '_MEDIT')", ;
        , "not skpbar('_MEDIT', _MED_CLEAR)")
    .Add('', '', '\-')
    .Add('', '', 'Select \<All', ;
        "sys(1500, '_MED_SLCTA', '_MEDIT')", ;
        , "not skpbar('_MEDIT', _MED_SLCTA)")
endwith
loMenu.Show()
```

As another example, the code in the RightClick method of TestPopMenu.SCX creates the relatively plain looking menu shown in Figure 2.

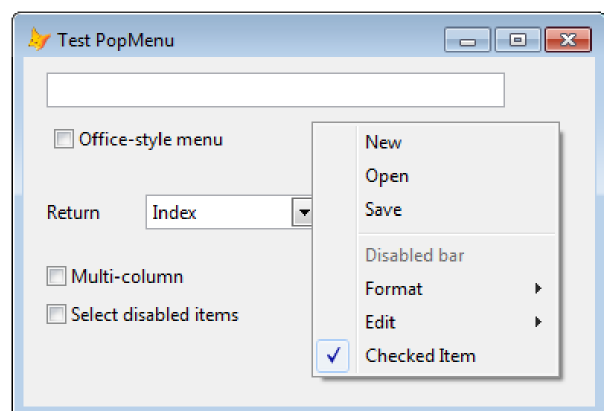

Figure 2. PopMenu can create object-oriented menus.

Getting fancier

So far, other than having an object-oriented menu, PopMenu hasn't done much for us yet. However, let's look at some properties that make menus really pop (pun intended).

The key to having more modern looking menus is setting lOwnerDraw to .T. This provides much better control over the appearance. The menu shown in **Figure 3** has this property set to .T. and a few other properties discussed later set so the menu has an Office-like appearance.

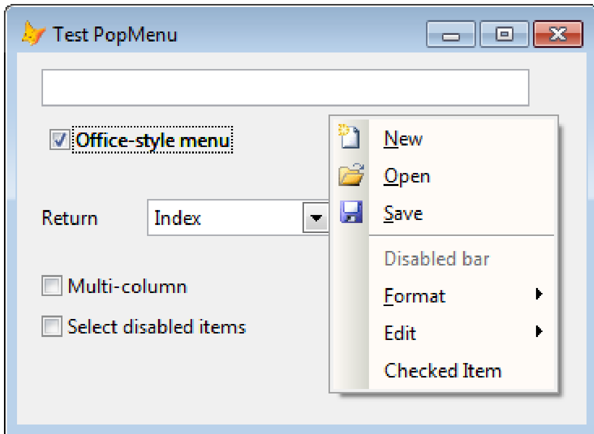

Figure 3. Setting lOwnerDraw to .T. is the key to more modern looking menus.

PopMenu has a few properties that control the dimensions of the menu items, as shown in **Figure 4**.

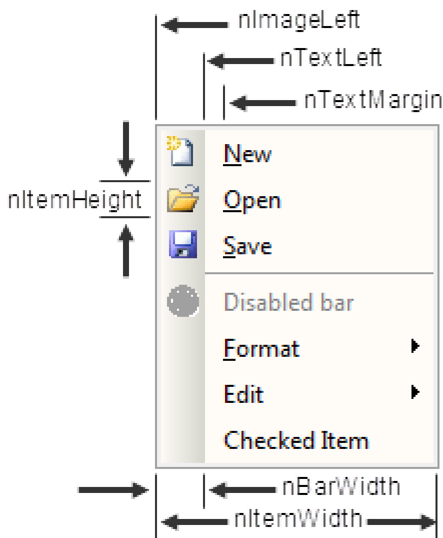

Figure 4. PopMenu has properties controlling the dimensions of menu items.

I created a subclass of PopMenu called SFOfficeMenu (in SFPopMenu.VCX) that has several properties set to create the menu shown in **Figure 3**. It has lOwnerDraw set to .T., nBarWidth set to 22, nItemHeight and nTextLeft set to 23, and nTextMargin set to 8. Other properties are set as described below. I created a subclass of SFOfficeMenu called SFEditMenu that provides a pre-defined menu with Cut, Copy, Paste, Clear, and Select All menu items.

nMenuBackColor is the color for the background of the menu items. It defaults to -1, which means use the system color for menus. SFOfficeMenu has this property set to RGB(255, 251, 247).

nBarStyle controls the appearance of the bar to the left of the menu items. The values are:

- 0: the bar doesn't have a separate appearance from the menu items.
- 1: fills the bar with the color specified in nBarFillColor1.
- 2: fills the bar with a left-to-right horizontal gradient from nBarFillColor1 (RGB(255, 251, 247) in SFOfficeMenu) to nBarFillColor2 (RGB(214, 215, 206) in SFOfficeMenu). SFOfficeMenu has nBarStyle set to 2.
- 3: fills the bar with a vertical gradient from nBarFillColor2 (top) to nBarFillColor1 (bottom).

If lSelectedEnabled is .F. (the default), a selection bar appears when you move over disabled items. Set this to .T. to display no bar.

If lShareIcons is .T. (the default), images for menu items are cached in a collection that's shared between all menus.

nSelectedForeColor determines the color of the text for the selected menu item; the default of -1 means use the system highlight text color. SFOfficeMenu has this set to RGB(0, 0, 0).

nSelectedStyle affects how the selected menu item appears. A related property is nSelectedImageStyle, which affects how the image for the selected menu item appears. These properties have the following values:

- 0: a solid bar of the color specified in nSelectedBackColor (the default of -1 means use the system highlight color) appears over the selected menu item. If nSelectedImageStyle is 0, the image is included in the same bar or rectangle as the text of the menu item; otherwise, it appears in a separate rectangle.
- 1: a rectangle appears over the selected menu item. The border of the rectangle uses nSelectedBorderColor as its color (the default of -1 means use the system highlight color). The fill color is specified in nSelectedBackColor if it isn't the default of -1; if it is the default, the fill color is a blend of nSelectedBorderColor (or the system highlight color if it's -1) and nMenuBackColor

(or the system menu color if it's -1). SFOfficeMenu has nSelectedStyle set to 1.

- 2: same as 1 but uses a rounded rectangle with nSelectedRoundX and nSelectedRoundY specifying the roundedness of the rectangle (they both default to 12).
- 3: displays a raised rectangle over the selected menu item. The fill color is specified in nSelectedBackColor if it isn't the default of -1; if it is the default, the system highlight color is used.
- 4: like 3, but displays a sunken rectangle.

If the user closes the menu without selecting an item, Show returns .NULL. nReturn determines what Show returns when the user selects an item:

- 0 (the default): return the index of the selected menu item.
- 1: return the value of the cKey property for the selected item.
- 2: return the title of the selected item.
- 3: returns an object reference to the selected item.

Show can accept four optional parameters. The first two are X and Y coordinates for the menu; if they aren't specified, the menu is positioned at the mouse location. Pass .F. for the third parameter if the X and Y coordinates are relative to the VFP window or .T. if they're absolute values (that is, relative to the screen). The last parameter is a numeric additive flag indicating how the menu should appear. Use one of the following values to specify how to position the menu horizontally:

- 0x4 (constant TPM_CENTERALIGN in Win32API.H): centers the menu horizontally.
- 0x0 (TPM_LEFTALIGN): positions the menu so its left side is at the X coordinate.
- 0x8 (TPM_RIGHTALIGN): positions the menu so its right side is at the X coordinate.

Use one of the following values to specify how to position the menu vertically:

- 0x20 (TPM_BOTTOMALIGN): positions the menu so its bottom is at the Y coordinate.
- 0x0 (TPM_TOPALIGN): positions the menu so its top is at the Y coordinate.
- 0x10 (TPM_VCENTERALIGN): centers the menu vertically.

Use one of the following values to specify which mouse button can be used:

- 0x0 (TPM_LEFTBUTTON): only the left mouse button can select items.
- 0x2 (TPM_RIGHTBUTTON): the user can use both the left and right buttons.

There are several values (and associated constants) that provide animation of the menu, but as with ctl32_ContextMenu, I couldn't get these to work.

You can also display the menu by calling ShowBy(oObject [, tnAddX [, tnAddY]]), where oObject is a reference to an object whose upper-left corner is used as the location of the menu and tnAddX and tnAddY are offset values to add to the X and Y values.

I fixed a few issues in PopMenu; see the comments in Init, CreateContext, CreateMenus, and Show for details.

To try out PopMenu, run the TestPopMenu form. Right-click in the text box to see a VFP shortcut menu, then turn on "Office-style menu" to use PopMenu instead. Right-click the form; it displays either a plain menu ("Office-style menu" turned off) or an Office-like menu (that setting turned on). Try turning on "Multi-column" and "Select disabled items" to see the effect they have.

What about system menus?

We've seen that PopMenu can make your shortcut menus look more modern. However, what about system menus? Unfortunately, that's a little more difficult to do. PopMenu uses the Windows API TrackPopupMenu function, which displays a shortcut menu. Shortcut menus are almost identical to popup menus (the menu that appears under a pad in a menu bar) except for one thing: they're modal. You have to click off a shortcut menu to close it without making a selection. A popup menu, on the other hand, closes when you move the mouse off it.

There's one other issue: where to display the menu. With shortcut menus, you expect the menu to appear at the mouse location. However, popup menus should appear directly under the menu pad, regardless of where the mouse is on that pad.

I created a sample program, TestPopMenu.PRG, that shows how to use PopMenu with a system menu bar. This program defines a menu bar using DEFINE PAD functions and uses ON SELECTION PAD to call functions that use PopMenu to display a menu. The vertical location for the menu can be determined using some SYSMETRIC() functions:

```
lnY = _vfp.Top  + sysmetric(9) + ;
```

```
    sysmetric(4) + sysmetric(20)
```

The complication is that the horizontal location of the menu isn't the location of the mouse but the location of the pad, which can't really be determined easily. For example, the location for the Tools pad uses a empirically determined offset of 40 to determine where the menu should go:

```
lnX = _vfp.Left + sysmetric(3) + 40
```

An easier solution is to dispense with the VFP menuing system and implement your own. I created a couple of classes in SFPopMenu.VCX, SFMenubar and SFPadButton, that provide a menu bar and pads as a proof of concept. TestSystemMenu.SCX uses these classes for a simple menu in a form.

Regardless of which mechanism you use, you can't escape the fact that the menus are modal, so the user can't slide the mouse from one pad to the next once they've clicked a pad; they have to click on the next pad to close the current popup and open the next one.

The Windows API also contains functions for dealing with system menu bars, so hopefully LingFeng Shi or someone else will implement them to allow us to deal with menus more gracefully.

Summary

PopMenu provides an easy way to add object-oriented shortcut menus that look like menus in modern applications such as Microsoft Office. However, it needs more work if you want to do something similar for system menu bars.

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of "Making Sense of Sedna and SP2," the "What's New in Visual FoxPro" series (the latest being "What's New in Nine"), "Visual FoxPro Best Practices For The Next Ten Years," and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He has been a Microsoft Most Valuable Professional (MVP) since 1996. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).