TransSkel Programmer's Notes

10: Button Outlining

Who to blame: Paul DuBois, dubois@primate.wisc.edu
Note creation date: 01/18/94
Note revision: 1.00
Last revision date: 03/24/95
TransSkel release: 3.07

This Note describes how to use TransSkel routines to manage the heavy outline that indicates default buttons.

03/24/95 — Minor edits.

Pushbutton controls occur in a variety of contexts in Macintosh programming, expecially during dialog or alert processing. It's standard practice to indicate the default button, if there is one, by a heavy outline surrounding the button. The outline indicates not only that the button is the default, but also that the Return and Enter keys are mapped onto clicks in the button. When the user types one of these keys, the button is flashed briefly. For example, in Figure 1, the Save button is indicated as the default by the heavy outline. The user can select the button by clicking it or by typing Return or Enter.

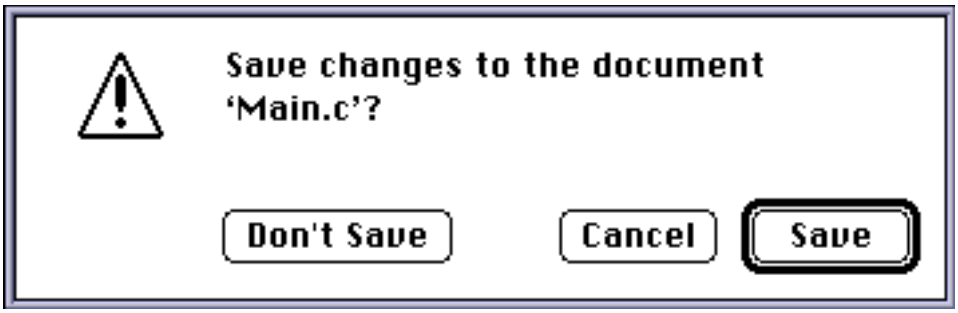

Figure 1. Dialog with Save button as default

The Macintosh Toolbox provides partial support for outlining the default button. If you call a function that presents an alert, the outline is drawn for you automatically. But if you call `ModalDialog()` or present a non-modal window containing buttons, you need to

take care of outlining the default button yourself. The Toolbox routine `SetDialogDefaultItem()` allows you to specify which button is the default for a modeless dialog and also causes the button to be outlined, but you can't use it unless you're running under System 7.

Generally, your application needs to do the following to perform button outlining:

- Draw the outline when you draw the button.

- Redraw the outline to match the button's state whenever the button changes from active to inactive and vice-versa.

- Erase the outline if you hide the button.

TransSkel provides a set of routines that make button outlining easier. To draw an outline around a button control, pass the control handle to `SkelDrawButtonOutline()`. This routine draws outlines with the following characteristics:

- Outlines are three pixels wide, separated from the button by one pixel.
- Outlines are drawn to match the hiliting state of the button: black if the button is active, gray if the button is inactive. For inactive buttons, the outline is drawn in true gray is possible. Otherwise, dithered gray is used.

To erase an outline around a button, pass the handle to `SkelEraseButtonOutline()`.

The following sections describe how to use TransSkel to outline buttons in different situations.

Outlining in Document Windows

To handle button outlining in a document window, include something like this in the code you use to respond to update events:

```
DrawControls (wind);
SkelDrawButtonOutline (buttonCtrl);
```

To respond to an activate event, set the button's activation state properly and invalidate the rectangle containing the outline to generate an update event so the outline will be redrawn. The code fragment below shows how to do this. It assumes that you make the default button active or inactive whenever the document window becomes active or inactive:

```
HiliteControl (buttonControl, comingActive ? 0 : 255);
r = (**buttonControl).contrlRect;
InsetRect (&r, -4, -4);
InvalRect (&r);
```

If you don't mind violating the "don't draw except in response to updates" principle, you can do this instead:

```
HiliteControl (buttonControl, comingActive ? 0 : 255);
SkelDrawButtonOutline (buttonCtrl);
```

Outlining in Dialogs

Outlining a button in a dialog requires a different treatment. You can't just call `SkelDrawButtonOutline()` any time you feel like it. For instance, during dialog updates, drawing is visible only within dialog item bounding rectangles. This means the

outline needs to be an item. The usual way of handling this problem is to include a user item in the dialog item list, position the item's bounding rectangle so that it surrounds the default button's bounding rectangle, and associate an outline-drawing procedure with the item. When the user item's rectangle is invalid, `ModalDialog()` calls the procedure associated with the item to draw the outline. (Note that since you need to set up the user item after creating the dialog, the dialog should be created invisible and then made visible with `ShowWindow()` after it's been set up.)

The easiest situation to handle is that in which the default button is the item specified in the dialog template as the default. In this case, call `SkelSetDlogButtonInstaller()` to take care of all the details for you. This function moves and sizes the user item bounding rectangle so it properly surrounds the default button, and associates the item with a drawing procedure that calls `SkelDrawButtonOutline()`. Here's an example:

```
dlog = GetNewDialog (dlogRsrcNum, nil, (WindowPtr) -1L);
GetPort (&tmpPort);
SetPort (dlog);
SkelSetDlogButtonInstaller (dlog, outlineItem);
ShowWindow (dlog);
for (;;)
{
	ModalDialog (nil, &item);
	if (item == okayItem || item == cancelItem)
		break;
}
DisposeDialog (dlog);
SetPort (tmpPort);
```

`SkelSetDlogButtonInstaller()` is sufficient by itself if your default button is always active. However, if you change the button's state from active to inactive or vice-versa, you should redraw the outline so it matches the button. Here's an example `ModalDialog()` loop that checks whether or not an edit text item contains anything and makes the state of an OK button and its outline active or inactive accordingly:

```
for (;;)
{
	ModalDialog (nil, &item);
	if (item == okayItem || item == cancelItem)
		break;
	/* set hiliting according to contents of edittext item */
	SkelGetDlogStr (dlog, editTextItem, str);
	hilite = (str[0] > 0 ? 0 : 255);
	if (SkelSetDlogCtlHilite (dlog, okayItem , hilite))
		SkelDrawButtonOutline (SkelGetDlogCtl (d, okayItem ));
}
```

`SkelSetDlogCtlHilite()` is helpful for avoiding redrawing. It does nothing and returns `false` if the control is already set properly. Otherwise it sets the hiliting value and returns `true`. The outline needs redrawing only when the return value is `true`. (Note: this example requires that the edittext item be enabled, so that `ModalDialog()` returns when the text is modified.)

`SkelSetDlogButtonOutliner()` works only for the button indicated as the default in the dialog template. If you want to associate the outline with an arbitrary button, you can't use `SkelSetDlogButtonOutliner()` and you must do some extra work yourself.

You still need a user item for the outline, but you must place and size the item's rectangle and associate the item with a drawing procedure. The drawing procedure is simple:

```
static pascal void
OutlineButton(DialogPtr d, short item)
{
	SkelDrawButtonOutline (SkelGetDlogCtl (d, defaultButton));
}
```

`defaultButton` indicates the item number of the default button. You have to maintain it as a global variable. Don't use the `item` parameter that's passed to the drawing routine because that's the number of the outline item, not the button item.

When you set up the user item, you calculate for the item bounding rectangle the same rectangle that `SkelDrawButtonOutline()` will use when outlining the button:

```
SkelGetDlogRect (dlog, defaultButton, &r);
InsetRect (&r, -4, -4);
SkelSetDlogRect (dlog, outlineItem, &r);
SkelSetDlogProc (dlog, outlineItem, OutlineButton);
```

Why associate the outline with a button other than that specified in the dialog template as the default? One reason is that your default button might change. Suppose you're presenting a dialog containing an edittext item allowing the user to specify a string to search for in a document. You may want to make the Cancel button the default when the edittext item is empty, and make the Find button the default otherwise. (See Figures 2 and 3.)

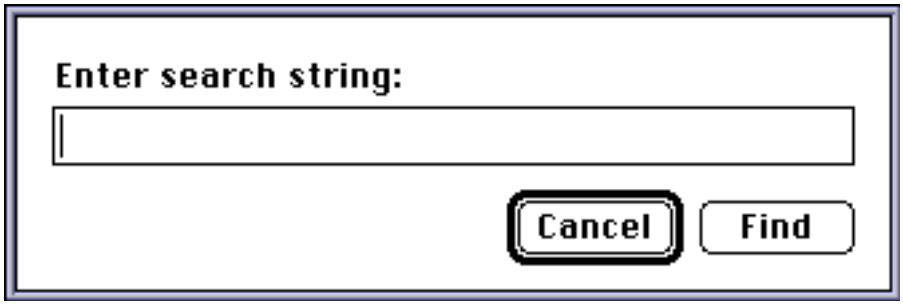

Figure 2. Dialog with Cancel button as default

Figure 3. Dialog with Find button as default

In a dialog like this, the item number of the default button can change. When this happens, you need to erase the outline from the button with which it's currently associated, and associate the outline item with the new default.

Assuming the initial search string is empty, set up Cancel as the initial default button:

```
SetDefaultButton (cancelItem);
```

`SetDefaultButton()` looks like this:

```
void
SetDefaultButton (DialogPtr dlog, short item)
```

```
{
Rect   r;

       defaultButton = item;
       SkelGetDlogRect (dlog, defaultButton, &r);
       InsetRect (&r, -4, -4);
       SkelSetDlogRect (dlog, outlineItem, &r);
       SkelSetDlogProc (dlog, outlineItem, OutlineButton);
       SkelDrawButtonOutline (SkelGetDlogCtl (dlog, defaultButton));
}
```

To change the default button, do something like this in your `ModalDialog()` loop:

```
for (;;)
{
       ModalDialog (nil, &item);
       if (item == okayItem || item == cancelItem)
              break;
       /* set default item according to contents of edittext item */
       SkelGetDlogStr (dlog, editTextItem, str);
       newDefault = (str[0] > 0 ? findItem : cancelItem);
       if (newDefault != defaultButton)
       {
              SkelEraseButtonOutline (SkelGetDlogCtl (dlog, defaultButton));
              SetDefaultButton (dlog, newDefault);
       }
}
```

When the default changes, the loop erases the current outline and sets up the new default button.

When you have a dynamic default button, there's an additional complication, which manifests itself as a bug in the preceding `ModalDialog()` loop. If you pass a `nil` filter function to `ModalDialog()`, it maps the Return and Enter keys onto dialog item 1. That's incorrect behavior if the default button isn't item 1. This problem can be avoided by using TransSkel's standard dialog filter mechanism and specifying the default item explicitly. For good measure, specify the cancel item, too, so that Escape and Command-period map to the Cancel button:

```
for (;;)
{
       filter = SkelDlogFilter (nil, false);
       SkelDlogDefaultItem (defaultButton);
       SkelDlogCancelItem (cancelItem);
       ModalDialog (filter, &item);
       SkelRmveDlogFilter ();
       if (item == okayItem || item == cancelItem)
              break;
       /* set default item according to contents of edittext item */
       SkelGetDlogStr (dlog, editTextItem, str);
       newDefault = (str[0] > 0 ? findItem : cancelItem);
       if (newDefault != defaultButton)
       {
              SkelEraseButtonOutline (SkelGetDlogCtl (dlog, defaultButton));
              SetDefaultButton (dlog, newDefault);
       }
}
```

User Item Caveats

When you create a user item for outlining a button in a dialog, you should observe the following constraints:

- The item must be disabled. Otherwise `ModalDialog()` returns the item number when you click in the item.

- The item should appear later in the item list than any enabled items, and it *must* be later in the item list than any button with which it can be associated. This is because the Dialog Manager apparently determines which item is clicked in by running through the item list until it finds an item with a bounding rectangle containing the click. If the item is disabled, the click is ignored, otherwise the item number is returned. Therefore, if the user item appears in the item list before the button with which it is associated, clicks in the button are interpreted as clicks in the user item and ignored.

Getting True Gray Outlines in Dialogs

If you create a dialog with `GetNewDialog()` and it is possible for the default button to become inactive, create a `'dctb'` resource for the dialog. This will allow the outline to be drawn in true gray on systems and monitors that support color or grayscale. Otherwise dithered gray will always be used.

You will also get dithered gray if you create a dialog with `NewDialog()`. To get true gray when possible, test whether Color QuickDraw is available, and use `NewCDialog()` instead if it is. You can do this as follows:

```
if (SkelQuery (skelQHasColorQD))
       dlog = NewCDialog ( ... arguments ... );
else
       dlog = NewDialog ( ... arguments ... );
```