Photon Control Surfaces #3: Putting Surfaces to Work

by David W. LeBlanc

Welcome to the third and final installment devoted to a species of neglected little creatures in the QNX Photon library that we like to call control surfaces.

In the first installment, we discussed control surfaces in general and how you might use them. We also looked at the control surfaces API. Then, in the second installment, we answered some common questions about how control surfaces are implemented and why the API is the way it is.

In this final installment, we dig below the surface and look at how to make these creatures work for us. To illustrate, we'll use keypad.c, a small application that implements a simple numeric keypad using only one widget - a **PtWindow**! (This wasn't possible in previous versions of Photon unless you either used **PtRaw** widgets or did your own drawing outside of the normal event handling loop.)

To build the example, download the source and run the command:

```
cc keypad.c -o keypad -l ph
```

Function 1: main()

As in other C programs, the heart of our example resides in *main()*, which initializes a Photon window, creates control surfaces to implement the buttons and character display area, and then enters the Photon main loop, which allows an application to interact with the Photon environment.

Let's start with the arguments to **PtCreateSurface**(). The first surface we create is for the numeric display field that goes along the top of the window. Here, we've assigned a known ID to the surface (**DISPLAY_SURFACE_ID**) so that we can easily reference it again. We could have let the library pick an ID by passing 0 for the ID argument (which in general is the best thing to do, since the operation will fail if the ID is already taken), but in this case we're pretty much guaranteed that our desired ID is free. Also, this way is clearer for the purpose of learning.

We then tell the library that our surface is rectangular and ask it to allocate a points array for us - the API was designed with laziness in mind :-). Since we don't want our display field to react to events, we set the event mask and callback both to 0. A fullblown example might look at key events and accept numerical input, but I'll leave this as an exercise for the reader. Finally, we set our draw and geometry calculation function pointers to **display_draw**() and **display_calc**(), respectively.

© 2003, 2007, QNX Software Systems. All rights reserved.

Next, we need to construct the buttons that make up the keypad. I've chosen to neglect '0' because it would've complicated the layout and thus obscured the example. To do this, I've used a simple "for" loop, which follows the first **PtCreateSurface**() call.

In subsequent calls to **PtCreateSurface**, we pass in our loop variable as the numerical ID. I've chosen to do this here because it makes it easier to determine which number the button represents later on in the callback functions. We could've done this in other ways (via **PtSurfaceAddData**() for instance), but this quick and easy mechanism works well for this case.

We've chosen to make the buttons elliptical and have again asked the library to allocate the points for us. The buttons will be sensitive to mouse button presses and releases, so we set our event callback, draw, and geometry calculation functions to **button_callback**(), **button_draw**(), and **button_calc**(), respectively.

Function 2: button_callback()

This is where we implement a button's action. A button typically behaves as follows:

- On the press, the button simply redraws itself in a pressed state.
- After the subsequent release, the button needs to redraw itself, whether or not the release occurred inside the button (fortunately, phantom button releases make this easy). If the release actually occurred inside the button, perform the action ("real" releases are suitable for this purpose).

To indicate a enabled/disabled state, we use surface data. The convention we've adopted here is that if no data is present, the button isn't pressed. If data is present (the data doesn't actually have to point to anything), the button is pressed. So, to indicate that the button is pressed, we use **PtSurfaceAddData**(), passing ~0 as the argument (any non-NULL value will do) and 0 as the **data_length** so that no copying is performed. To indicate that the button is no longer pressed, we use **PtSurfaceRemoveData**(). After changing the button's state, we need to call **PtDamageSurface**() so that the surface redraws itself in its new state.

Function 3: button_draw()

This function draws the button. To get the color scheme, we retrieve the fill and foreground colors of the parent widget (**PtWindow**). If the button is pressed (disclosed by the call to **PtSurfaceGetData**()), we invert the color scheme.

Next, we draw an ellipse and then inside we draw our number, which is discovered simply by retrieving the numerical ID of the surface (as discussed above) in the **main**() section.

Function 4: button_calc()

This function calculates the button's bounding box. The function gets called for each button, so using the ID, we figure the row/column position of the button. Then, knowing the size of the **PtWindow**, we figure out the position and size that the surface should be using. The code to do this is just boring math; nothing magical.

Function 5: display_draw()

This function draws the numerical text field. We've chosen to store the buffer in the display surface's data, which is a more typical use for surface data, as opposed to buttons using their data to store their enabled/disabled state.

© 2003, 2007, QNX Software Systems. All rights reserved.

Function 6: display_calc()

As with **button_calc**(), we calculate geometry only after the widget has calculated its own geometry (which is disclosed via the post argument passed in to the function). The remainder of the geometry calculation is uninteresting, so I'll leave it as an exercise for you to review this code.

Function 7: add_to_display()

This function appends a number to the text buffer and is called from within the button callbacks when a complete button press-release cycle occurs (i.e. a release within the pressed button). Most of the code here isn't surface-specific, aside from how the surface data is dealt with.

We first make a call to **PtFindSurface**() to retrieve a pointer to the structure describing the surface associated with numerical **ID DISPLAY_SURFACE_ID**. We don't have to do this. We could have used the **ById** forms of the API throughout, but they're a bit slower since they have to do the same lookup each time. So as an optimization step, we do the lookup just once and use the direct forms of the calls thereafter. (Using the surface pointer is safe until we add or remove surfaces from the widget, in which case the pointer might become invalid.)

Next, we retrieve the surface data. For the first time through, this ought to return NULL, since it hasn't been set yet. So, in this case, we allocate a buffer on the stack and add it as surface data. It's okay to do this since the function will make a copy of the data that we've passed it. In fact, this is why we made the second call to **PtSurfaceGetData**(), which might otherwise seem redundant. Remember that since the buffer has been copied, we need a pointer to the surface's copy, not to our own (which in this case will soon become invalid once the function returns and our stack frame is gone).

After we've appended our character to the buffer (scrolling it if necessary), we call **PtDamageSurface**()to get the display field to update itself.

It's not a bug, it's a feature...

Compile the sample. Run it. Play with it. Resize the window. Oops! What a mess. Change consoles and back again. Ahh, that's better! Hm, must be a bug... or is it?

As you may remember, the example from the first article in this series exhibited similar behavior. This isn't a bug. As we discussed in the first article, control surfaces lack some of the library services provided to widgets. The artifact we're seeing here is simply a service "deficiency" designed into the API to make surfaces as economical as possible. Under most circumstances, widgets don't get damaged (redrawn) automatically when they're resized, since they typically don't need to. Their contents (usually just a flat, filled rectangle) don't vary with their size. The widgets don't know, nor do they try to know, about their control surfaces or the content of those surfaces (which in this case depends on the size of the window).

Thus it's up to the application developer to make provisions for these shortcomings. And in this particular case, the fix is simple. When the window is resized, it simply needs to redraw itself fully. This requires the addition of a one-line callback function - not including the return statement - on the window. I'll leave the rest up to you. If you find yourself **really** stuck, then cheat!

Taking control

Remember that you can add control surfaces to any widget. For instance, you could add a magical hot spot to a **PtButton**. Or install an "event hole" into a widget to disable or override a portion of it. Don't like the way **PtCombobox** works by default? No problem! Control surfaces let you safely hack away at a widget to your heart's content. Remember that if you return **Pt_END** in your callbacks - or merely use the

© 2003, 2007, QNX Software Systems. All rights reserved.

Pt_SURFACE_CONSUME_EVENTS flag in **PtCreateSurface**() - it'll be like the event never happened, as far as the widget itself is concerned. Keep in mind, however, that filter callbacks do get the first crack at incoming events.

Have fun!

© 2003, 2007, QNX Software Systems. All rights reserved.