Control surfaces #2: FAQ

by David W. LeBlanc

Welcome to installment #2 of the ongoing series of articles devoted to our tiny little friends hidden away in the deep recesses of the Photon library -- control surfaces!

In the first article of this series, you were introduced to control surfaces and what they can do for you. After doing some homework and perusing the API, you might be left with more questions than answers. This is normal. Although we intended the control surface API to parallel the rest of the Photon library as closely as possible, we needed to diverge from the mainstream in some instances to achieve our design goals.

So in an attempt to quench that fiery thirst for answers that some of you may be experiencing after having read the first article, I've decided to shape this article into a Q&A format to answer some of the questions I get asked frequently on this topic.

Q. What is the proper way to reference a control surface?

A. Confusion on this issue reigns because there are two ways to reference control surfaces. One is by a direct pointer to the control surface structure (**`PtSurface_t *`**). The other is by a numerical identifier (16-bit unsigned **PtSurfaceId_t**). While the pointer method is more direct and therefore quicker, it's not as safe as the ID method. To understand why, we need to be aware of how control surfaces are organized and stored in memory.

Unlike the widget hierarchy, which is implemented as a linked list, control surfaces are stored as an array of surface structures (**PtSurface_t**). There are a couple of major reasons for storing them in this manner:

1. The array allows for quick traversal in both directions (which is a requirement, since drawing is handled from back to front and events are processed from front to back).

2. The array reduces the memory requirement per surface. To satisfy the quick-traversal requirement, a doubly linked list would have to be used. Otherwise, we'd need an additional 8 bytes per surface, not to mention the overhead of the memory allocation itself for each surface. While 8 to 16 additional bytes per surface may not sound like much overhead, when compared to the size of the surface itself (24 bytes), this amounts to a minimum 50% increase in size!

Surface addition and removal were deemed sufficiently low-bandwidth operations to make this approach feasible without too much of a performance penalty.

So armed with this knowledge we see that as control surfaces are physically moved around in the stacking order, their placement in the array will change, affecting their address in memory. In addition, as surfaces are added or removed to/from a widget, the array needs to be reallocated, which also may cause the array itself to move around in memory. With all this possibility of memory movement, numerical identifiers are the only reliable way of locating a surface.

That being said, if you're pretty certain that a widget's surface configuration isn't going to change, then the pointer method is safe (and a heck of a lot quicker, since the ID method needs to do a linear lookup in the surface array).

Q. How/when do I calculate my surfaces' geometry?

A. Your surfaces will be asked to calculate their geometry twice when the widget that owns them is asked to calculate its geometry ("extented" as the legacy terminology would put it):

1. once before the widget's geometry calculation (which allows a widget to size itself according to the requirements of its surfaces if it cares -- and some widgets do)

2. once after (allowing surfaces to position and size themselves according to the size of the widget).

In the latter case, the post argument will be non-zero -- as an application designer, this is probably the only case you'll ever care about.

A surface may also calculate its geometry based on the geometry of other surfaces. Using **PtCalcSurface[ById]**, you can ensure that the surface you're interested in has calculated its geometry prior to examining it.

The actual recording of the surface's geometry is simply a matter of directly modifying the surface's points array. Be sure you're aware of how this array is organized before proceeding. This organization is detailed in the documentation for **PtCreateSurface**() from the first article.

Q. How are control surfaces drawn?

A. Control surfaces are asked to draw themselves from back to front, after the widget itself has drawn. No clipping is done for you. If you want clipping, you'll have to implement the necessary logic to adjust the clipping list as surfaces are traversed, and then reinstate the clipping stack after the last surface is drawn. Otherwise, you'll see some very interesting (and probably undesirable) visual artifacts.

Q. There's no callback data in the callback functions! How do I pass data to my callback functions?

A. There's no callback data, because that data would have to be stored somewhere. And for what we deemed to be a relatively unnecessary feature, we weren't prepared to sacrifice an additional 4 bytes per surface. Use **PtSurfaceAddData[ById]** and **PtSurfaceGetData[ById]** to bind data to a surface.

Q. Dang! These surfaces are so tight. There's no room to move around!

A. Hey, that wasn't a question! Yes, the surfaces and the API are somewhat "packed" and might feel reminiscent of the restrooms on an airplane. But like any design engineer from any aircraft manufacturer will probably tell you, this compactness is there by design. Of paramount concern in the implementation was to keep these things as lean as possible, and we believe we've done that.

Be thankful -- in the original cut, control surfaces were (if you can believe this) 16 bytes apiece, which isn't a whole lot when you consider that the function pointers themselves took up 12. But after some additional feature requests and API reviews, surfaces grew into the unwieldy beasts they are today.

Q. Any other tips?

A. Don't do a **while**(1); in your callbacks. :) But you probably already knew that.