Subject: Oddball Event Handling (Longer than it Ought to Be)
Posted by davidf on Sun, 31 Dec 2000 04:46:43 GMT
View Forum Message <> Reply to Message

Folks,

Speaking of oddball event handling (We were speaking about oddball event handling, weren't we?), I've been fooling around with an interesting project where I am building a bunch of compound widgets (out of objects, naturally). These widgets can show up in various incarnations, sometimes as part of another widget program, sometimes in their own top-level base, etc.

The event handler for the compound widget is simple, it just reads a "message" stored in the user value of each widget that can cause an event, and the message tells the event handler what method to call to handle the event (via a Call_Method command). That part is simple, simple, simple, and you can read all about it in my book. :-)

But, when the event is through being processed there is a possibility that some other widget, which is NOT part of the compound widget, might want to know about it. For example, the compound widget might change the character size of a plot, and the widget that is re-drawing the plot might want to know about the character size thingy being changed, etc.

This is absolutely no problem if the compound widget is part of the widget hierarchy of the plot re-draw widget, but it *was* a problem if the compound widget happened to be in its own top-level base. What would happen is that the event I wanted to send would have the wrong top-level base ID, thereby causing the info structure with all the program information for the other widget program to become lost.

Oh, man. I hate this kind of problem!

I sometimes solve it by putting the info structure in a pointer and passing the pointer here, there, and everywhere. But, uughhh. That involves a *bunch* of modifications, and then I have to explain this program to the client, and I've been talking about simpler is better, and .... Well, you get the idea.

By the way, you aren't doing anything else today, are you?
I mean, it being the first of the year and all. Because
I haven't even gotten to the *point* of this article yet. :-)

So, I got to thinking that I wanted something that was
simpler than modifying my whole info structure scheme.
What I wanted to know was the identifier of that *other*
top-level base over there, so I could substitute it's
identifier in event.top for the one I had, before I passed
the event along. And what I knew was the identifier of
the parent of the compound widget, which just happened
to *always* be in the hierarchy of that widget over there.

(OK, here comes the point of this too long story.)

All I had to do was traverse UP the widget hierarchy
until I got to the top!

Now, you have to understand, I din't have no computer
learnin in schol. Recursive functions and I have no
common ground. But a recursive function is what I needed.
(Can something that goes UP be recursive!?)

So, anyway, after fooling around for 10 minutes I
wrote something that--somehow, for some reason known
only to JD and a couple of others, probably--works!!!!

I don't have a clue. Really. But I thought it might
be useful for y'all, in case you ever wanted to do
something like this. You just pass it a widget ID, and
it spits out the widget identifier at the top of that
widget hierarchy.

Pretty neat, huh? I *love* IDL sometimes. :-)

Happy New Year,

David

```
 ********************************************************** ***
FUNCTION FindTLB, startID

; This function traces up the widget hierarchy to find the top-level base.

FORWARD_FUNCTION FindTLB
parent = Widget_Info(startID, /Parent)
IF parent EQ 0 THEN RETURN, startID ELSE parent = FindTLB(parent)
RETURN, parent
```

END
********** ******

P.S. Let's just say I don't really want to hear about it
if this thing doesn't work. It works for me. :-)

--
David Fanning, Ph.D.
Fanning Software Consulting
Phone: 970-221-0438 E-Mail: davidf@dfanning.com
Coyote's Guide to IDL Programming: http://www.dfanning.com/
Toll-Free IDL Book Orders: 1-888-461-0155

Subject: Re: Oddball Event Handling (Longer than it Ought to Be)
Posted by Michael Plonski on Sun, 07 Jan 2001 02:29:25 GMT
View Forum Message <> Reply to Message

I had played around with an object based widget system a while back, but then I moved on to other things. The basic design was an object wrapper for any well behaved widget (one that left user value for the user). THe user value was used to store object information. Each widget had both an ow (object widget) parent and a regular widget parent and they did not have to be the same (useful if you just wanted a widget parent for formatting the widget on the screen but not for functionality). The base object wrapper had all the neccessary methods so that events could be assigned to object methods, by revectoring the normal widget event handler. I built a series of basic ow, including a generic gui object that had an image ow, status lines ow, toolbar ow, etc. Each of these componets was its own ow, so that you could inherit the ow and then add new capabilities to it. This is very hard to do with regular widget unless you cut and paste and then rewrite the widget functions. It had a nice feature that when you revectored the events, it stored the state of the object widget. The basic design was that an image object widget should not respond to anything other than window events - resize, redraw, etc.. When a user selected something like draw a line from one of the toolbar ow, it would ask the gui ow for the ow pointer of the image ow, and then execute a method on the image ow to revector the mouse click events. The method for drawing lines was in the toolbar widget where it belongs and not in the image window. For example there was a different toolbar button for a constrained line draw, so that the line could only be drawn at a specific angle. The image window had no knowledge of these operations which is what made it reusable with any toolbar ow. The image ow would automatically push its state onto a stack, when it revectored the event handler. This let the event be revectored later, like only respond to clicks and not motion, by some other drawing object widget. As each ow completed, it would execute a method on the image ow to return event controller which would then pop

the previous state from the stack. The design was kind of nice in that an image_ow had no event handlers for dealing with mouse clicks, since a generic image display shouldn't deal with mouse. It is applications that use the image ow that deal with the mouse. These applications were in effect, toolbar ow that could be added into the gui ow so that you could reuse the generic image services of an image ow. As you added ow toolbars into the gui, they would override the image ow events when they were active. Similiarly, if they wanted to report status, they would ask the gui ow for an object pointer to a status ow, and then send their status comment to that object. THis made for very modular gui development. What was really nice is that since the widgets were now objects you could inherit from them and add functionality. FOr example the base object widget wrapper had a generic method to handle event so that they would not go untrapped. After you inherited from this ow, you would override the event handler to be what you needed to make a generic toolbar widget. A genric toolbar widget could then be inherited to make a specific toolbar widget. What was nice in the object widget approach is that you would inherit new features. Initially, I had only designed the revector event handler method to save the current state. Later on I found it useful to push the state on the stack, so that you could revector a revectored event and still roll back to the initial state. Changinf this is the base object widget wrapper propagated to all events since this is the base class for all later inheritence. Since there is a parallel widget tree and object tree, destroying either the top level object or widget would destroy both the widget and object tree. The base level object widget wrapper took care of these kinds of things so that all object widget that inherited from it would fit within the parent tree structure. No need to go and write what happens when a widget is destroyed for each individual widget since you now just inherit this functionality I built a working application to demostrate that the whole infrastructure worked and it has been extremely reliable, no dangling widgets or objects after creating and deleteing guis. I just thought I give you a little input if you are going to start down the same path of making object widgets.

Well my 3 year old just came down and wants to bump me off the computer so he can play computer games. That shows you my priorities. I hope the above is readable since I can't review it with my 3 year bumping me off the machine.

Mike Plonski

David Fanning wrote:
>
> Folks,
>

> Speaking of oddball event handling (We were speaking about
> oddball event handling, weren't we?), I've been fooling
> around with an interesting project where I am building
> a bunch of compound widgets (out of objects, naturally).
> These widgets can show up in various incarnations,
> sometimes as part of another widget program, sometimes
> in their own top-level base, etc.
>
> The event handler for the compound widget is simple,
> it just reads a "message" stored in the user value of
> each widget that can cause an event, and the message
> tells the event handler what method to call to handle
> the event (via a Call_Method command). That part is
> simple, simple, simple, and you can read all about it
> in my book. :-)
>
> But, when the event is through being processed there
> is a possibility that some other widget, which is NOT
> part of the compound widget, might want to know about it.
> For example, the compound widget might change the
> character size of a plot, and the widget that is re-drawing
> the plot might want to know about the character size thingy
> being changed, etc.
>
> This is absolutely no problem if the compound widget
> is part of the widget hierarchy of the plot re-draw
> widget, but it *was* a problem if the compound widget
> happened to be in its own top-level base. What would
> happen is that the event I wanted to send would have
> the wrong top-level base ID, thereby causing the
> info structure with all the program information for the
> other widget program to become lost.
>
> Oh, man. I hate this kind of problem!
>
> I sometimes solve it by putting the info structure in
> a pointer and passing the pointer here, there, and
> everywhere. But, uughhh. That involves a *bunch* of
> modifications, and then I have to explain this program
> to the client, and I've been talking about simpler is
> better, and .... Well, you get the idea.
>
> By the way, you aren't doing anything else today, are you?
> I mean, it being the first of the year and all. Because
> I haven't even gotten to the *point* of this article yet. :-)
>
> So, I got to thinking that I wanted something that was
> simpler than modifying my whole info structure scheme.

> What I wanted to know was the identifier of that *other*
> top-level base over there, so I could substitute it's
> identifier in event.top for the one I had, before I passed
> the event along. And what I knew was the identifier of
> the parent of the compound widget, which just happened
> to *always* be in the hierarchy of that widget over there.
>
> (OK, here comes the point of this too long story.)
>
> All I had to do was traverse UP the widget hierarchy
> until I got to the top!
>
> Now, you have to understand, I din't have no computer
> learnin in schol. Recursive functions and I have no
> common ground. But a recursive function is what I needed.
> (Can something that goes UP be recursive!?)
>
> So, anyway, after fooling around for 10 minutes I
> wrote something that--somehow, for some reason known
> only to JD and a couple of others, probably--works!!!!
>
> I don't have a clue. Really. But I thought it might
> be useful for y'all, in case you ever wanted to do
> something like this. You just pass it a widget ID, and
> it spits out the widget identifier at the top of that
> widget hierarchy.
>
> Pretty neat, huh? I *love* IDL sometimes. :-)
>
> Happy New Year,
>
> David
>
> ** ***
> FUNCTION FindTLB, startID
>
> ; This function traces up the widget hierarchy to find the top-level base.
>
> FORWARD_FUNCTION FindTLB
> parent = Widget_Info(startID, /Parent)
> IF parent EQ 0 THEN RETURN, startID ELSE parent = FindTLB(parent)
> RETURN, parent
> END
> ** ******
>
> P.S. Let's just say I don't really want to hear about it
> if this thing doesn't work. It works for me. :-)
>

> --
> David Fanning, Ph.D.
> Fanning Software Consulting
> Phone: 970-221-0438 E-Mail: davidf@dfanning.com
> Coyote's Guide to IDL Programming: http://www.dfanning.com/
> Toll-Free IDL Book Orders: 1-888-461-0155

Subject: Re: Oddball Event Handling (Longer than it Ought to Be)
Posted by davidf on Sun, 07 Jan 2001 02:40:48 GMT
View Forum Message <> Reply to Message

Michael Plonski (mplonski@aer.com) writes:

> I had played around with an object based widget system a while back, but
> then I moved on to other things. The basic design was an object wrapper
> for any well behaved widget (one that left user value for the user).
> THe user value was used to store object information. Each widget had
> both an ow (object widget) parent and a regular widget parent and they
> did not have to be the same (useful if you just wanted a widget parent
> for formatting the widget on the screen but not for functionality). The
> base object wrapper had all the neccessary methods so that events could
> be assigned to object methods, by revectoring the normal widget event
> handler. I built a series of basic ow, including a generic gui object
> that had an image ow, status lines ow, toolbar ow, etc. Each of these
> componets was its own ow, so that you could inherit the ow and then add
> new capabilities to it. This is very hard to do with regular widget
> unless you cut and paste and then rewrite the widget functions. It had
> a nice feature that when you revectored the events, it stored the state
> of the object widget. The basic design was that an image object widget
> should not respond to anything other than window events - resize,
> redraw, etc.. When a user selected something like draw a line from one
> of the toolbar ow, it would ask the gui ow for the ow pointer of the
> image ow, and then execute a method on the image ow to revector the
> mouse click events. The method for drawing lines was in the toolbar
> widget where it belongs and not in the image window. For example there
> was a different toolbar button for a constrained line draw, so that the
> line could only be drawn at a specific angle. The image window had no
> knowledge of these operations which is what made it reusable with any
> toolbar ow. The image ow would automatically push its state onto a
> stack, when it revectored the event handler. This let the event be
> revectored later, like only respond to clicks and not motion, by some
> other drawing object widget. As each ow completed, it would execute a
> method on the image ow to return event controller which would then pop
> the previous state from the stack. The design was kind of nice in that
> an image_ow had no event handlers for dealing with mouse clicks, since a
> generic image display shouldn't deal with mouse. It is applications
> that use the image ow that deal with the mouse. These applications were

> in effect, toolbar ow that could be added into the gui ow so that you
> could reuse the generic image services of an image ow. As you added ow
> toolbars into the gui, they would override the image ow events when they
> were active. Similiarly, if they wanted to report status, they would
> ask the gui ow for an object pointer to a status ow, and then send their
> status comment to that object. THis made for very modular gui
> development. What was really nice is that since the widgets were now
> objects you could inherit from them and add functionality. FOr example
> the base object widget wrapper had a generic method to handle event so
> that they would not go untrapped. After you inherited from this ow, you
> would override the event handler to be what you needed to make a generic
> toolbar widget. A genric toolbar widget could then be inherited to make
> a specific toolbar widget. What was nice in the object widget approach
> is that you would inherit new features. Initially, I had only designed
> the revector event handler method to save the current state. Later on I
> found it useful to push the state on the stack, so that you could
> revector a revectored event and still roll back to the initial state.
> Changinf this is the base object widget wrapper propagated to all events
> since this is the base class for all later inheritence. Since there is
> a parallel widget tree and object tree, destroying either the top level
> object or widget would destroy both the widget and object tree. The
> base level object widget wrapper took care of these kinds of things so
> that all object widget that inherited from it would fit within the
> parent tree structure. No need to go and write what happens when a
> widget is destroyed for each individual widget since you now just
> inherit this functionality I built a working application to demostrate
> that the whole infrastructure worked and it has been extremely reliable,
> no dangling widgets or objects after creating and deleteing guis. I
> just thought I give you a little input if you are going to start down
> the same path of making object widgets.

I can't wait to read the book! :^)

Cheers,

David

P.S. Let's just say I thought James Joyce's Ulysses
was a hard read.

--
David Fanning, Ph.D.
Fanning Software Consulting
Phone: 970-221-0438 E-Mail: davidf@dfanning.com
Coyote's Guide to IDL Programming: http://www.dfanning.com/
Toll-Free IDL Book Orders: 1-888-461-0155