Subject: Re: widget_control and group_leader
Posted by John-David T. Smith on Sun, 24 Dec 2000 19:48:08 GMT
View Forum Message <> Reply to Message

Nidhi Kalra wrote:
> Let me paste in a bit of the code from program A. In the event handler
> that I am mostly concerned with, the user sets the mode. mousemode
> cases 0-3 were already there, so I added 4 for uniformity. When 4 is
> selected, events on the draw_widget are sent to the foreign event
> handler.
>
> pro a_event, event
> ; Main event loop for atv top-level base, and for all the buttons.
> ...
> widget_control, event.id, get_uvalue = uvalue
> ...
> case uvalue of
> 'mode': case event.index of
> 0: widget_control, state.draw_widget_id, $
> event_pro = 'atv_draw_color_event'
> 1: widget_control, state.draw_widget_id, $
> event_pro = 'atv_draw_zoom_event'
> 2: widget_control, state.draw_widget_id, $
> event_pro = 'atv_draw_blink_event'
> 3: widget_control, state.draw_widget_id, $
> event_pro = 'atv_draw_phot_event'
> 4: widget_control, state.draw_widget_id, $
> event_pro = state.foreign_event_handler $
> + '_event'
> else: print, 'Unknown mouse mode!'
> endcase

This design choice of A (and A's author probably is scratching his head about now), is simply one possiblity among the zillions of ways events flow can be managed. He has decided to redirect events to different procedures, based on which "button" or mode is selected. Naturally, you added yet another mode to this. I guess I haven't looked closely enough at A to speak sensibly about its design choice. But the one point I wanted to make was that there is no need for events to be processed only in 1 place. That is, you could easily have zoom, blink, phot, or color active at the same time B is receiving these event! Wouldn't this be better functionality, unless something B is doing is really making A's default behavior non-intuitive.

>
> The functionality I'm going for is that the user can decide when to use
> external event handlers and when to let program A run 'naturally'. At
> the moment, I have tried to keep foreign_event as general as possible.
> Each B can do whatever it pleases with its own particular foreign_event

> handler. The two things (now) registered with A are the foregn event
> handler to use and a widget_ID to use. Whatever that needs to be.
>
>> What I would recommend in this case is set up a foreign event handler
>> *method*, since the foreign widget is an object. That is, have a
>> routine to sign up for events from A. from within B., like this:
>>
>> a_signup, self, "Handle_A_Events", /Button, /TRACKING
>>
>> or some such. Then, each "foreign" object can sign up for whatever
>> events it wants.
>

> How do I register event/object pairs? Ok. So here I'm a little lost
> (Caution: Newbie IDL-er at work). A
> registers the following event handlers:
>
> widget_control, top_menu, event_pro = 'topmenu_event'
> widget_control, state.draw_widget_id, event_pro = 'draw_color_event'
> widget_control, state.draw_base_id, event_pro = 'draw_base_event'
> widget_control, state.keyboard_text_id, event_pro = 'keyboard_event'
> widget_control, state.pan_widget_id, event_pro = 'pan_event'
>
> And everything in these main bases is differentiated by uvalues (as you
> can see from the above code). So I'm a bit confused about how to go
> about differentiating the "events requested" and how the reigstering
> in "call_method,method,obj, ev" works.

Sorry if I didn't give enough detail. You register it however you want! You don't need to use IDL's builtin event_pro stuff, and in many cases, it's more convenient not to. To make things concrete, consider that you might like A to remain unmodified. You'd make an event handler app "C" (I'd do it as an object):

```
pro Broker::signup, obj, method, REMOVE=rm
  if obj_valid(obj) eq 0 then return
  if ptr_valid(self.signup) then begin
     wh=where((*self.signup).object eq obj,cnt,COMPLEMENT=valid)

     ;; Rid list of obj, if it's already on there
     if cnt ne 0 then begin
        if valid[0] eq -1 then ptr_free, self.signup $
        else *self.signup=(*self.signup)[valid]
     endif
  endif

  ;; Add it to the list, if necessary
  if keyword_set(rm) then return
```

```
  list_item={BROKER_SIGNUP,Object:obj,Method:Method}
  if ptr_valid(self.signup) then begin ; append item
     *self.signup=[*self.signup, list_item]
  endif else $                  ;create list with item
     self.signup=ptr_new(list_item,/NO_COPY)
end

pro broker_handler, ev
  widget_control,ev.top, get_uvalue=self
  self->Handler
end

pro Broker::Handler, ev
  ;; Send it to A
  widget_control, self.A_ID, SEND_EVENT=ev

  ;; Send it to all the B's
  if ptr_valid(self.signup) eq 0 then return

  for i=0,n_elements(*self.signup)-1 do begin
     call_method, (*self.signup)[i].Method, $
                  (*self.signup)[i].Object, ev
  endfor
end

pro Broker::Cleanup
  ptr_free,self.signup
end

pro Broker::Init, A_ID
  ;; We will pre-process A's events
  widget_control,A_ID, event_pro= 'c_handler'
end

pro Broker__Define
  struct={Broker,A_ID: A_ID, signup:ptr_new(), ...}

  ;; A convenience struct for the signup list
  list_struct={BROKER_SIGNUP, object:obj_new(), method:''}
end
```

This is just an outline, and I haven't tried it. But it gives you an idea of what I had in mind. You can see how I caught A's events before it does, and then send them on to A (via the standard IDL event flow), and also to the B's (via the method/object they signed up for). You could obviously add more intelligence to this dispatch process (e.g. only button events to B1, etc.)

This uses IDL's widget hierarchy some, and some of it's own design too (e.g. the B methods). This is no problem. An event is simply a structure, no different from any other type of data, and you can use it however you want.

> So, ideally, here's the functionality im looking for. On "foreign"
> mode, all events go to foreign_event_handler. If foreign event handler
> wants to do something with it, wonderful. If not, the event goes back
> to where it would go on non-foreign mode.

So, the one thing I didn't specify is when the B's signup for events. I.e. how do they know they are on? Two possibilities:

1. They are always on, i.e. you start them from the command line, and they immediately sign up:

```
a=A_widget(lots_of_args)
c=obj_new('Broker',a)
b=obj_new('Cool_foreign_helper',BROKER=c)
```

and in b's Init:

```
function Cool_foreign_helper::Init, BROKER=brk
  brk->Signup, self, 'MyHandler'
  return, 1
end
```

2. They get turned on by A (which means you'd have to modify A to at least have this ability).

> The quick and dirty way is to put in a simple statement in each of the
> four event handlers:
>
> if (foreign) send_event, foreign_event_handler, event (or whatever).
>
> hmm...waitaminit. what if i register foreign_event_handler as the event
> handler for the top level base? what would that do? Would all events
> then go to foreign_event_handler and then bubble up/down?

These kind of uncertainties about just how IDL will handle dynamically re-routed events are just the thing that motivates moving beyond the standard event flow paradigm. What I gave is only a sketch, but once you take events "into your own hands", you can accomplish all sorts of things with them.

Remember, events are data too, just like 4 or "a string". They can be used to control widgets, and all you have to have are the widget_id's to make this work. Just because RSI publishes a manual describing standard event processing

doesn't mean you can't innovate beyond that (especially in unusual cases like yours).

Good luck,

JD
