Combining Objects and Compound Widgets

Rob Dimeo
July 20, 2004

The following excerpt is from the on-line help for IDL:

A compound widget is a complete, self-contained, reusable widget sub-tree that behaves to a large degree just like a widget primitive, but which is written in the IDL language. Compound widgets allow the development of reusable widget code, much like a GUI subroutine.

Widget Values of Compound Widgets

Many compound widgets have associated values. Initial values can often be specified using the VALUE keyword to the creation routine. Note, however, that in some cases widget values of compound widgets cannot be set until after the widget is realized; values are thus set, obtained, or changed using the GET_VALUE and SET_VALUE keywords to the WIDGET_CONTROL procedure. See the documentation for the individual compound widget creation routines in the IDL Reference Guide for more detailed information.

Limitations of Conventional Compound Widgets

Often compound widgets (cw) are written in a manner similar to those in the IDL distribution such as CW_BGROUP, CW_FIELD, etc. We will refer to this style as conventional. Conventional compound widgets do not use objects in their implementation. As stated above, users can extract and modify the widget values from conventional compound widgets using the GET_VALUE and SET_VALUE keywords to WIDGET_CONTROL. However there is no straightforward and general method to modify the appearance of a compound widget after it has been *realized*. Often the user of a cw (i.e. developer of code that uses a cw) would like to be able to change the appearance of some aspect of it such as the text in a title portion of the CW.

In order to circumvent this inherent limitation, it is possible to exploit the properties of objects. A cw function interface can be written in such a way as to wrap an object widget that it will be called in nearly the exact same manner as a conventional cw but offers more flexibility and control than a conventional cw. In order to understand how to exploit the capability of objects for this purpose we need to review the requirements of a conventional cw.

Before continuing, it is assumed that most readers are comfortable with the "standard" way of writing a conventional IDL widget program in which the state information is passed around between event handlers using the UVALUE of the top-level base (TLB).

Requirements and common practices for conventional compound widgets

-"state" information must be stored in location other than the UVALUE of the top-level base of the cw. This is required since the UVALUE of the TLB must be available to the user of the cw.

-"state" information is typically stored in the UVALUE of the first child of the cw's top-level base. In the IDL distribution the first child of the top-level base is called the *stash*. Storing the state in the stash as such makes retrieval in an event handler easy requiring only a call such as the following:

```
stash = widget_info(event.top,/child)
widget_control,stash,get_uvalue = state
```

-There is no need to call XMANAGER in the code for the cw since the cw will be managed by the XMANAGER called in the code of the cw's parent.

-Since there is no XMANAGER call in the CW, there can be no *cleanup* routine as is usually recommended in a conventional widget program. Rather one has to implement a KILL_NOTIFY procedure and associate it with the STASH.

-The event handler for the cw (i.e. the *internal* event handler) must be set for the STASH.

-In order to allow the functionality of getting and setting the value of the cw, it is necessary to set the keywords PRO_SET_VALUE and FUNC_GET_VALUE to the names of the associated procedure and function in the call to WIDGET_BASE that defines the cw's top-level base.

-The compound widget must be written as a function that returns the widget id of the cw's top-level base.

-The compound widget should be written such that its functional interface contains all of the usual fields (parameters and keywords) in a standard primitive widget (e.g. WIDGET_BUTTON). Such fields include, but are not limited to, the following:

PARENT	(parameter)
EVENT_FUNC	(keyword)
EVENT_PRO	(keyword)
UNAME	(keyword)
VALUE	(keyword)
UVALUE	(keyword)

The interface for a compound widget object

A conventional compound widget, such as CW_FIELD, is called from a parent widget program as follows:

```
id =  cw_field(tlb,title = 'My widget', value = 'My value')
```

Extracting the value of the cw is simply a matter of making the following procedure call:

```
widget_control,id,get_value = cw_text_val
```

The punchline to this entire method of writing cw's as objects is that you can make the following call (to an imaginary compound widget object named CWO_FIELD):

```
id=cwo_field(tlb,title='My widget',value='My value',obj_ref=o)
```

The call is nearly identical to that of CW_FIELD. However the last keyword in this function call is the object reference to the underlying widget object. With this object reference available to the programmer, it is possible to invoke any of the underlying methods on the widget object. One example might be to change the title of the widget (which is impossible with CW_FIELD). The way you might change the title might be through an accessor method for the object named set_property as follows:

```
ret = o->set_property(title = 'New Title')
```

Let's take a detailed look at a concrete implementation of a compound widget object.

RMD_DICE: A compound object widget replacement for CW_DICE

An example of a cw is provided in the IDL distribution and documentation called CW_DICE.PRO and you can examine the code for the compound widget.

```
IDL> .edit cw_dice
```

As you examine the code in CW_DICE you will notice that it suffers from a few deficiencies. First and foremost the code uses common blocks to store the faces of the die and the random number seed used when rolling the die. This can lead to problems if you use more than one instance of the cw when conflicts can arise between "common variables". Of course it is not difficult to rewrite this code with a minimal amount of effort to eliminate the COMMON

block by simply storing these values in the STASH. However we wish to make the code more flexible by implementing it as an object widget.

In contrast to a procedural program an object posesses attributes and behavior. The attributes are data members. Therefore it is simple to store the required state information (widget or otherwise) for an object widget as an attribute. Moreover the behavior of an object is defined by the object's methods. These methods enable modification of the widget's appearance or any other property of the widget, which leads us to the second deficiency in the conventional cw program CW_DICE.

The second flaw in CW_DICE as written is that it is impossible to modify any property of the die except the value after it has been realized in its parent widget. For example, the keyword TUMBLE_CNT specifies the number of faces you will see when you roll the die before it shows the final resulting face. This can only be set upon creating the cw. By writing this as an object widget we can extract the object reference (o) in the call to the function and then invoke the accessor method SET_PROPERTY as follows:

```
ret = o->set_property(tumble_cnt = 5)
```

Therefore it should be clear that there are advantages in writing cw's as objects. In order to understand how to write a cw as an object, let's take a look at RMD_DICE, a drop-in replacement for CW_DICE.

RMD_DICE is composed of the following procedures and functions:

RMD_DICE	(F)
RMD_DICE__DEFINE	(P)
RMD_DICE::INIT	(F)
RMD_DICE::INIT_DIE_FACES	(F)
RMD_DICE::CONVERT_BMP	(F)
RMD_DICE::DICE_EVENT_HANDLER	(F)
RMD_DICE::ROLL	(F)
RMD_DICE::TOSS	(F)
RMD_DICE::GET_PROPERTY	(F)
RMD_DICE::SET_PROPERTY	(F)
RMD_DICE::CLEANUP	(P)
RMD_DICE_SET_VAL	(P)
RMD_DICE_GET_VAL	(F)
RMD_DICE_KILL_NOTIFY	(P)
RMD_DICE_EVENTS	(F)

A short description of the purpose of each of the procedures is listed below.

RMD_DICE (F)

Wrapper function for an instance of the object class RMD_DICE. This provides the same interface to the object widget as the interface to CW_DICE.

RMD_DICE__DEFINE (P)

Definition module for the class called RMD_DICE. The types for the data needed in the object, the attributes, are defined here.

RMD_DICE::INIT (F)

Initialization module. This is one of two lifecycle methods in the class definition. In this module the object data are initialized and the widgets are defined.

RMD_DICE::INIT_DIE_FACES (F)

Loads bitmaps of the six die faces stored on disk.

RMD_DICE::CONVERT_BMP (F)

Converts the format of the bitmaps to one that IDL displays nicely and resizes the images.

RMD_DICE::DICE_EVENT_HANDLER (F)

This is the event handler for the object. It returns an event structure to the calling program (i.e. the parent widget) composed of ID, TOP, HANDLER, VALUE, and OBJECT. It also starts a die roll by invoking the method RMD_DICE::ROLL. This also calls the external event handler (function or procedure) as specified by the user in the inital function call to RMD_DICE:

e.g. `id = rmd_dice(tlb,obj_ref = o,event_pro = 'my_dice_event')`

RMD_DICE_EVENTS (F)

Dispatches widget events to the proper method via RMD_DICE::DICE_EVENT_HANDLER.

RMD_DICE::ROLL (F)

Displays a tumbling die and calls RMD_DICE::TOSS to obtain a random result.

RMD_DICE::TOSS (F)

Uses random number generator to simulate die toss.

RMD_DICE::GET_PROPERTY (F)

Accessor method.

RMD_DICE::SET_PROPERTY (F)
Accessor method.

RMD_DICE::CLEANUP (P)
Frees heap variables (data members) of the object.

RMD_DICE_SET_VAL (P)
Required procedure for setting the value of the CW via
`widget_control,id,set_value = new_value.`

RMD_DICE_GET_VAL (F)
Required function for retrieving the value of the CW via
`widget_control,id,get_value = current_value.`

RMD_DICE_KILL_NOTIFY (P)
"Cleanup" routine associated with the stash that destroys the object.

Using RMD_DICE in a sample program

There is an example program named DICE_EXAMPLE.PRO appended to the program RMD_DICE.PRO. This program demonstrates how to incorporate the cw named RMD_DICE into a widget program. It has four components:

DICE_EXAMPLE (P)
Widget definition module. Note that both the NO_PRESS and NO_CNTDOWN keywords are set in the function call to RMD_DICE. This is "by-design" so that the user cannot press the die and make it roll.

DICE_EXAMPLE_EVENT (P)
Event handler for all events from widget. Handles events pertaining to rolling once, rolling N times, printing the current value of the die, and quitting.

ROLL_ONCE (P)
Invokes WIDGET_CONTROL statement to roll the die once. There are two ways to roll the die. One way is to set the value of the die widget to -1. The other is to invoke the ROLL method directly on the object widget. Both methods are included in the source code with one commented out.

DE_CLEANUP (P)
Eliminates the pixmap and clears any heap variables.

In order to run the program, compile RMD_DICE and type DICE_EXAMPLE at the command line:

```
IDL> .compile RMD_DICE
IDL> DICE_EXAMPLE
```

You should see a user-interface like that shown in fig. 1.

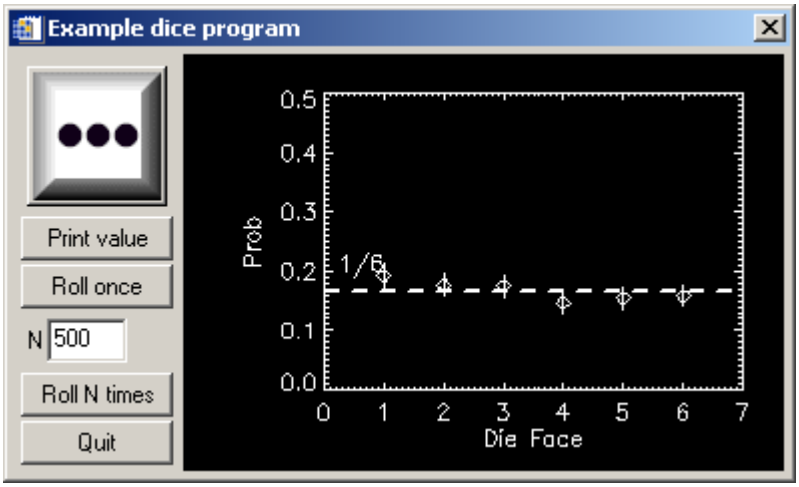

Fig. 1 Screen shot from DICE_EXAMPLE.PRO.

Chuck-a-Luck Simulator

The program XCHUCK_A_LUCK is a conventional widget program that uses the RMD_DICE compound widget for all of the die needs. We will discuss the rules of the game first and then briefly discuss the odds of winning and the expected return on this game of chance.

The game of Chuck-A-Luck is an old favorite and can be found still in traveling carnivals. The rules are simple and the odds *appear* to be favorable for the player. As we will see, the odds are naturally in favor of the house. The game apparatus consists of three dice and a betting surface has regions containing the numbers 1 through 6 corresponding to the faces of a die. The bettor places money in one of the six regions on the betting surface and the the three dice are rolled simultaneously. For the purposes of illustration let's assume that the bet is $1. If the number does not appear on any of the three dice then the stakes are lost. If the number appears once then the bettor wins $1 plus the $1 bet. If the number appears twice then the bettor wins $2 plus the $1 bet. If the number appears all three times then the bettor wins $3 plus the $1 bet.

The program XCHUCK_A_LUCK.PRO simulates this game of chance in IDL. The application is shown in fig. 2.

Fig. 2 Screen shot of XCHUCK_A_LUCK.

In this application you begin with a purse of $100. You can bet any amount up to $100. You can also select which die face in the field labeled *Dice number*. When you have entered your bet and selected the *Dice number* then you begin playing by pressing the button labeled *Roll Dice*. The faces on the three dice will change for a few moments and then the result will be displayed. The fields labeled *overall winnings*, *winnings this roll*, and *purse* will be updated based on the outcome of this roll.

It is easy to figure out how likely a bettor is to win any return. In order to understand this we can consider the probabilities.

Consider rolling one die. This can result in only 2 outcomes. Let X represent the event in which this single die matches the bettor's choice. Let Y represent the event in which there is no match between the bettor's choice and the die. With three dice there are eight (2^3) possible events:

XXX,XXY,XYX,XYY,YXX,YXY,YYX,YYY

We know of course that the probability of X occuring is P(X)=1/6. Similarly we know that P(Y)=5/6. Now we can enumerate the probabilities.

Probability of three matches (XXX):
$P(3\ matches) = P(XXX) = P(X) \bullet P(X) \bullet P(X) = 1/6^3 = 1/216$

Probability of two matches: (XXY,XYX,YXX)
$P(2\ matches) = P(XXY)+P(XYX)+P(YXX) = 3 \bullet (1/6^2) \bullet (5/6) = 15/216$

Probability of one match: (YYX,YXY,XYY)
$P(1\ match) = P(YYX)+P(YXY)+P(XYY) = 3 \bullet (5/6)^2 \bullet (1/6) = 75/216$

Probability of no matches: (YYY)
$P(0\ matches) = P(YYY) = (5/6)^3 = 125/216$

With this information we can determine the probability of winning on any single game.

$P(win) = P(3\ matches) + P(2\ matches) + P(1\ match) = 91/216 = 0.42$

Therefore the odds are clearly in favor of the house.

Now we can calculate the expected return on a $1 bet.

$Return = \$3 \bullet (1/216) + \$2 \bullet (15/216) + \$1 \bullet (75/216) - \$1 \bullet (125/216) = -\$0.08$.

For every dollar that you bet you can expect to pay the house 8 cents.

We can test out different payouts to see the effects. For instance, if we change the payout of a triple from 3:1 to 10:1 we find:

Return = \$10•(1/216) + \$2•(15/216) + \$1•(75/216) - \$1•(125/216) = -\$0.05.

You can continue increasing the odds of a triple match to find out when the odds are even. The result is a triple payout of 20:1.

Source Code Listing

```
; NAME:
;
;  RMD_DICE
;
; PURPOSE:
;
;  Demonstrate how to make a compound widget using an object
;  widget.  This particular example uses the functionality of
;  CW_DICE.PRO that comes in the IDL distribution.  It builds
;  on that functionality with a NO_PRESS keyword that allows
;  the developer to invoke die rolls programmatically.  In
;  addition it uses bitmapped images of the 6 faces of a die.
;  If the program cannot find the nice bitmaps of the die then
;  it uses the same ones that are in the CW_DICE routine.
;
;  Since this compound widget is written as an object, the
;  COMMON block in CW_DICE contained in the IDL distribution
;  has been eliminated.
;
; AUTHOR:
;
;  Robert Dimeo
;  National Institute of Standards and Technology
;  Center for Neutron Research
;  100 Bureau Drive, Mail Stop 8562
;  Gaithersburg, MD 20899
;  Tel: (301) 975-8135
;  Email: robert.dimeo@nist.gov
;
; CATEGORY:
;
;  General programming
;
; CALLING SEQUENCE:
;
;  DICE_ID = RMD_DICE(PARENT,VALUE = 1)   ; Sets the initial
;                                         ; value to 1
;
;  The return value of the RMD_DICE is the widget id of the new
;  widget, just like an ordinary primitive widget.
;
; INPUT PARAMETERS (required):
;
;  PARENT -- The id of the parent widget
;
; INPUT KEYWORDS (optional):
;
;  EVENT_FUNC --  Event handler function to be executed when
;                 event generated in the widget.  See EVENT
;                 STRUCTURE below for a description of the
;                 event structure returned.
;  EVENT_PRO --   Event handler procedure to be executed when
;                 event generated in the widget.  See EVENT
;                 STRUCTURE below for a description of the
;                 event structure returned.
;  UNAME --       The user name of the RMD_DICE widget
;  UVALUE --      The user value of the RMD_DICE widget
;  TUMBLE_CNT --  The widget simulates the tumbling of a die by
;                 changing the bitmap on the die several times
;                 before settling down to a final value.  The
;                 number of "tumbles" is specified by the
```

```
;                 TUMBLE_CNT keyword.  The default is 10.
;  TUMBLE_PERIOD --
;                 The amount of time in seconds between each
;                 tumble of the dice.  The default is 0.05 seconds.
;  NO_PRESS --    If set then pressing the die will not result in a
;                 a roll.  A roll can only be invoked programmatically
;                 by setting UVALUE = -1
;  NO_CNTDOWN --  Setting this keyword prevents the intermediate
;                 steps in a roll from being displayed.
;
; OUTPUT KEYWORDS (optional)
;
;  OBJ_REF --     Object reference to the compound widget allowing the
;                 developer to invoke methods directly on the widget
;                 and bypassing the limitations imposed by the
;                 WIDGET_CONTROL procedure.
;
; EVENT STRUCTURE:
;
;  event = {RMD_DICE_EVENT,   $ Name of event structure
;           ID:0L,            $ The ID of the compound widgets TLB
;           TOP: 0L,          $ The id of the TLB of the hierarchy
;           HANDLER:0L,       $ The event handler ID.
;           VALUE:0,          $ The value of the die (0<int val<7)
;           OBJECT:SELF}        The object reference allowing the user
;                               to call methods on the object directly.
;
; METHODS:
;
;  These accessor methods are used when the user invokes either of the
;  following two commands:
;
;     WIDGET_CONTROL, DICE_ID, GET_VALUE = VAL
;     WIDGET_CONTROL, DICE_ID, SET_VALUE = VAL
;
;  SET_PROPERTY --   Accessor method allowing user to change any of the
;                    "DATA" listed below.
;  GET_PROPERTY --   Accessor method allowing user to obtain any of the
;                    "DATA" listed below.
;  "DATA" --         VALUE, NO_CNTDOWN, NO_PRESS, FACES, TLB_ID
;   Note that TLB_ID is only available in GET_PROPERTY.
;
; COMMON BLOCKS:
;
;  NONE
;
; REQUIREMENTS:
;
;  SOURCEROOT.PRO
;  Bitmap files: d1.bmp, d2.bmp, ..., d6.bmp located in the directory
;  labeled IMAGES.

; PROCEDURE:
;
;  The RMD_DICE widget consists of a single pushbutton that displays
;  its current die value as a bitmap.  If the user presses the button,
;  it tumbles for a moment and then the new value is displayed and an
;  event is issued.  The tumbling effect is present only if the
;  TUMBLE_CNT keyword is set to a number greater than 1 and the
;  NO_CNTDOWN keywork is not set.
;
;  The current value of the die is available via the
;  WIDGET_CONTROL, DICE_ID, GET_VALUE = VAL command.
```

```
;
;  The current value of the die can be set by issuing the
;  WIDGET_CONTROL, DICE_ID, SET_VALUE = VAL command.  If the
;  requested value is outside the range [1,6], then the die
;  tumbles to a new value as if the user had pressed the
;  button, but no event is issued.
;
;  The die can be "rolled" programmatically by one of two methods.
;  The first case was presented in the paragraph above via
;
;     widget_control,dice_id,set_value = 0
;
;  The second way to roll the die programmatically is by invoking the
;  ROLL method on the object itself via
;
;     ret = o_die->roll()
;
; EXAMPLE:
;
;  An example implementation of this compound widget is appended
;  to the end of this program listing and it is called DICE_EXAMPLE.
;  Compile this program and then type DICE_EXAMPLE at the IDL prompt.
;
;  This example program allows the user to roll the die once, print
;  out the resulting value, or roll it many times and see a discrete
;  probability density function showing the frequency of occurrences
;  of each face.  A dashed line is superposed on the probability
;  density to indicate the value of 1/6, the theoretical value.
;
; MODIFICATION HISTORY:
;
;  Written -- 2/3/04 (RMD)
;  Fixed bug that occurred for multiple rolls -- 2/6/04 (RMD)
;  Added NO_PRESS keyword to prevent user from rolling a die.  This
;     facilitates the developer allowing only programmatic rolls.
;  Also added SEED as an object attribute.  SEED is used in the
;  random number generator for rolling the die.
;     -- 2/7/04 (RMD)
;  Added OBJ_REF keyword in the widget function call so that the
;  object reference can be passed out to the user enabling direct
;  invocation of methods on the widget object -- 2/23/04 (RMD)
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro rmd_dice::cleanup
ptr_free,self.storage,self.seed
heap_free,self.faces
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function rmd_dice::set_property,    value = value,              $
                                    no_cntdown = no_cntdown,    $
                                    no_press = no_press,        $
                                    tumble_cnt = tumble_cnt,    $
                                    faces = faces

if n_elements(tumble_cnt) ne 0 then self.tumble_cnt = tumble_cnt
if n_elements(value) ne 0 then self.value = value
if n_elements(no_cntdown) ne 0 then self.no_cntdown = no_cntdown
if n_elements(no_press) ne 0 then self.no_press = no_press
if n_elements(faces) ne 0 then *self.faces = faces
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function rmd_dice::get_property,    value = value,              $
```

```
                                    no_cntdown = no_cntdown,   $
                                    no_press = no_press,       $
                                    faces = faces,             $
                                    tlb_id = tlb_id

value = self.value
no_cntdown = self.no_cntdown
no_press = self.no_press
tlb_id = self.tlb
if arg_present(faces) then faces = *self.faces
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function rmd_dice_get_val,id
stash = widget_info(id,/child)
widget_control,stash,get_uvalue = object
ret = object->get_property(value = value)
return,value
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function rmd_dice::toss
if n_elements(*self.seed) gt 0 then $
   s = *self.seed
toss = fix(6*randomu(s)+1)
*self.seed = s
return,toss
end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function rmd_dice::roll
if self.no_cntdown then tc = 1 else tc = self.tumble_cnt
if self.remaining eq 0 then begin
   self.remaining = tc
   self.value = self->toss()
endif
if self.remaining eq 1 then begin
   value = self.value
endif else begin  ; self.remaining gt 1 or less than 0
   value = self->toss()
   widget_control,self.dice_id,timer = self.tumble_period
endelse
widget_control,self.dice_id,set_value = *(*self.faces)[value-1]
self.remaining = self.remaining - 1
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro rmd_dice_set_val,id,value
stash = widget_info(id,/child)
widget_control,stash,get_uvalue = object
if (value lt 1) or (value gt 6) then begin
   ret = object->roll()
endif else begin
   ret = object->set_property(value = value)
   ret = object->get_property(faces = faces)
   widget_control,stash,set_value = *faces[value-1]
endelse
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function rmd_dice::dice_event_handler,event
this_event = {RMD_DICE_EVENT, id:self.tlb,           $
                              top:event.top,         $
                              handler:event.handler, $
                              value:self.value,      $
```

```
                              object:self            }
in_event = tag_names(event,/structure_name)
if strupcase(in_event) eq 'WIDGET_BUTTON' then begin
   if self.no_press eq 1 then return,0
endif
; Call the event if requested
if self.event_pro ne "" then begin
   call_procedure,self.event_pro,this_event
   this_event = 0
endif
if self.event_func ne "" then begin
   ret = call_function(self.event_func,this_event)
   this_event = 0
endif

; Roll the die and display the results
ret = self->roll()

; If this invocation is not from the timer, issue an event
if tag_names(event,/structure_name) eq 'WIDGET_TIMER' then return,0
return,this_event
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function rmd_dice_events, event
widget_control,event.id,get_uvalue = object
the_event = object->dice_event_handler(event)
return,the_event
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro rmd_dice_kill_notify, wid
; This is the internal "cleanup" routine for the compound widget.
; It simply destroys the object.
widget_control,wid,get_uvalue = object
obj_destroy, object
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function rmd_dice::convert_bmp,filename
x = read_bmp(filename,r,g,b)
xsize = size(x)
dxsize = (dysize = 64)
if xsize[1] eq dxsize and xsize[2] eq dysize then xred = x else $
   xred = congrid(x,dxsize,dysize)
bmp = reform([[r[xred]],[g[xred]],[b[xred]]],dxsize,dysize,3)
return,bmp
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function rmd_dice::init_die_faces
; Load in the die face bitmaps and store them in a pointer to a
; pointer array (length 6)
path = sourceroot()+'images'+path_sep()
bmp_arr = ptrarr(6,/allocate_heap)
if file_test(path+'d1'+'.bmp') then begin
   for i = 1,6 do begin
      dice_file = path+'d'+strtrim(string(i),2)+'.bmp'
      d_bmp = self->convert_bmp(dice_file)
      *bmp_arr[i-1] = d_bmp
   endfor
endif else begin
   faces = lonarr(192)
   i4=indgen(4)+1
   s5=[0,5]
   pos=[13, 77, 141, 36, 54, 67, 87, 100, 118, 131, 151, 164, 182]
   v1=['c00300'x,'c00300'x,'c00300'x,'e0010000'x,'8007'x,   $
```

```
         'f0000000'x,'f'x, 'e0018007'x,'e0018007'x,'f000000f'x, $
         'f000000f'x,'f0c0030f'x,'f0c0030f'x]
   v2=['e00700'x,'e00700'x,'e00700'x,'f0030000'x,'c00f'x, $
         'f8010000'x,'801f'x, 'f003c00f'x,'f003c00f'x,'f801801f'x,   $
         'f801801f'x,'f8e1871f'x,'f8e1871f'x]
   for i = 0, n_elements(pos)-1 do begin
      faces[s5+pos[i]] = v1[i]
      faces[i4+pos[i]] = v2[i]
   endfor
   faces = byte(faces,0,4,32,6)
   BYTEORDER, faces, /HTONL     ; Little endian machines need swap
   for j = 1,6 do *bmp_arr[j-1] = faces[*,*,j-1]
endelse
ret = self->set_property(faces = bmp_arr)
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function rmd_dice::init,   parent,                          $
                           EVENT_FUNC = event_func,         $
                           EVENT_PRO = event_pro,           $
                           _Extra = extra,                  $
                           UNAME = uname,                   $
                           UVALUE = uvalue,                 $
                           VALUE = value,                   $
                           TUMBLE_CNT = tumble_cnt,         $
                           TUMBLE_PERIOD = tumble_period,$
                           NO_PRESS = no_press,             $
                           NO_CNTDOWN = no_cntdown

; Check for keywords that have been passed in
if n_params() eq 0 then return,0
if keyword_set(no_cntdown) then no_cntdown = 1B else $
   no_cntdown = 0B
if keyword_set(no_press) then no_press = 1B else no_press = 0B
if not (keyword_set(tumble_cnt)) then tumble_cnt = 10
if tumble_cnt lt 1 then tumble_cnt = 10
self.tumble_cnt = tumble_cnt
if not (keyword_set(tumble_period)) then tumble_period = 0.05
if tumble_period lt 0 then tumble_period = 0.05
self.tumble_period = tumble_period
if n_elements(uname) eq 0 then uname = ''
self.uname = uname
if n_elements(storage) eq 0 then storage = ''

self.no_cntdown = no_cntdown
; Initialize the die faces
self.faces = ptr_new(/allocate_heap)
ret = self->init_die_faces()
; Store the UVALUE if the user passed one in.
self.storage = ptr_new(uvalue)

; Use RANDOMU to pick the initial value of the die, unless the
; user provided one.
self.seed = ptr_new(/allocate_heap)
if n_elements(value) eq 0 then value = self->toss()
self.value = value
if n_elements(event_pro) eq 0 then event_pro = ""
if n_elements(event_func) eq 0 then event_func = ""

; Create the widgets
self.tlb = widget_base(parent,UVALUE = storage,    $
            FUNC_GET_VALUE = 'RMD_DICE_GET_VAL',   $
            PRO_SET_VALUE =  'RMD_DICE_SET_VAL',   $
```

```
            UNAME = uname, _Extra = extra          )

bmp = *(*self.faces)[value-1]
die = widget_button(self.tlb,value = bmp,  $
   UNAME = uname+'_BUTTON',kill_notify = 'RMD_DICE_KILL_NOTIFY',  $
   EVENT_FUNC = 'RMD_DICE_EVENTS', UVALUE = self,/bitmap)
self.dice_id = die
self.event_func = event_func
self.event_pro = event_pro
self.no_press = no_press
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro rmd_dice__define
define = {  rmd_dice,            $
            tlb:0L,              $
            dice_id:0L,          $
            no_press:0B,         $
            no_cntdown:0B,       $
            remaining:0,         $
            event_pro:'',        $
            event_func:'',       $
            tumble_cnt:0,        $
            tumble_period:0.0,   $
            value:0,             $
            uname:'',            $
            storage:ptr_new(),   $
            seed:ptr_new(),      $
            faces:ptr_new()      $
         }
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function rmd_dice,parent,_Extra = extra,OBJ_REF = obj_ref
; Provide an interface identical to that of CW_DICE.PRO and
; return the widget id.
obj_ref = obj_new('rmd_dice',parent,_Extra = extra)
ret = obj_ref->get_property(tlb_id = tlb_id)
return,tlb_id
end
;*****************************************************************
;************************Example usage*********************************
pro roll_once,event
; There are two ways to roll the die once.  The first one is
; commented out.  The second one is consistent with the way
; in which this is done using CW_DICE.

;widget_control,event.top,get_uvalue = pstate
;ret = (*pstate).o_die->roll()

id1 = widget_info(event.top,find_by_uname = 'DIE_1')
widget_control,id1,set_value = -1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro dice_example_event,event
widget_control,event.top,get_uvalue = pstate
uname = widget_info(event.id,/uname)
case uname of
'QUIT':  widget_control,event.top,/destroy
'PRINT_VALUE': $
   begin
      id = widget_info(event.top,find_by_uname = 'DIE_1')
      widget_control,id,get_value = this_val1
      print,this_val1
```

```
   end
'ROLL_ONCE': $
   begin
      ret = (*pstate).o_die->set_property(no_cntdown = 0)
      roll_once,event
      ret = (*pstate).o_die->set_property(no_cntdown = 1)
   end
'ROLL': $
   begin
      ret = (*pstate).o_die->set_property(/no_cntdown)
      id = widget_info(event.top,find_by_uname = 'NUM_ROLLS')
      widget_control,id,get_value = val & n = long(val[0])
      hist = fltarr(n)
      for i = 0,n-1 do begin
         roll_once,event
         id = widget_info(event.top,find_by_uname = 'DIE_1')
         widget_control,id,get_value = this_val1
         hist[i] = float(this_val1)
      endfor
      yhist = histogram(hist,min = 1.0,max = 6.0,nbins = 6)
      b = 1.0+findgen(6)

      y = yhist/(1.0*n)
      dy = sqrt(yhist)/(1.0*n)
      yth = 1./6.
      wset,(*pstate).winpix
      plot,b,y,psym = 4,yrange = [0.0,0.5],/ysty, $
         xrange = [0.0,7.0],/xsty,xtitle = 'Die Face',ytitle = 'Prob'
      errplot,b,y-dy,y+dy,width = 0.0
      plots,!x.crange,[yth,yth],/data,linestyle = 2,thick = 2.0
      xyouts,0.25,0.2,/data,'1/6'
      wset,(*pstate).winvis
      device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pstate).winpix]
      ret = (*pstate).o_die->set_property(no_cntdown = 0)
   end
else:
endcase
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro de_cleanup,tlb
widget_control,tlb,get_uvalue = pstate
wdelete,(*pstate).winpix
heap_free,pstate
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro dice_example
tlb = widget_base(/tlb_frame_attr,/row,title = 'Example dice program')
col_base = widget_base(tlb,/col)
row_base = widget_base(col_base,/row,/align_center)
tc = 20
tp = 0.01
void = rmd_dice(row_base,uname = 'DIE_1',tumble_count = tc, $
   tumble_period = tp,no_press=1,no_cntdown=1,obj_ref = o_die)

void = widget_button(col_base,value = 'Print value',uname =
'PRINT_VALUE')
void = widget_button(col_base,value = 'Roll once',uname = 'ROLL_ONCE')
void = cw_field(col_base,xsize = 5,value = '500',title = 'N',   $
   uname = 'NUM_ROLLS')
void = widget_button(col_base,value = 'Roll N times',uname = 'ROLL')
void = widget_button(col_base,value = 'Quit',uname = 'QUIT')
col_base_geom = widget_info(col_base,/geom)
xsize = 300 & ysize = col_base_geom.ysize
```

```
win = widget_draw(tlb,xsize = xsize,ysize = ysize,uname = 'WIN')

widget_control,tlb,/realize

widget_control,win,get_value = winvis
window,/free,/pixmap,xsize = xsize,ysize = ysize
winpix = !d.window

state = {ones:0L,winvis:winvis,winpix:winpix,o_die:o_die}
pstate = ptr_new(state,/no_copy)
widget_control,tlb,set_uvalue = pstate
xmanager,'dice_example',tlb,/no_block,cleanup = 'de_cleanup'
end
```

```
; NAME:
;  XCHUCK_A_LUCK
;
; PURPOSE:
;  Simulate a simple game of chuck-a-luck.
;
; CALLING SEQUENCE:
;  XCHUCK_A_LUCK
;
; INPUT PARAMETERS:
;  NONE
;
; KEYWORD PARAMETERS:
;  NONE
;
; REQUIREMENTS:
;  CENTERTLB.PRO
;  RMD_DICE.PRO -- a compound widget that supplants the
;  program CW_DICE.PRO.  This compound widget does not
;  use any COMMON blocks.
;
; COMMON BLOCKS:
;  NONE
;
; AUTHOR:
;  Rob Dimeo
;  February 3, 2004
;
; MODIFICATION HISTORY:
;  Changed fonts to larger, easier-to-read fonts
;  -- RMD (2/7/04)
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xcac_cleanup,tlb
widget_control,tlb,get_uvalue = pstate
heap_free,pstate
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xcac_get_values,event
widget_control,event.top,get_value = pstate
names = (*pstate).names
for i = 0,n_elements(names)-1 do begin
   dice_id = widget_info(event.top,find_by_uname = names[i])
   widget_control,dice_id,get_value = dice_value
   if i eq 0 then dice = dice_value else dice = [dice,dice_value]
endfor

end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function xcac_reset,event
widget_control,event.top,get_uvalue = pstate
(*pstate).purse = 100
(*pstate).winnings = 0
format = '(f8.2)'
purse_id = widget_info(event.top,find_by_uname = 'PURSE')
widget_control,purse_id,set_value = string((*pstate).purse,    $
   format = format)
win_id = widget_info(event.top,find_by_uname = 'WINNINGS')
widget_control,win_id,set_value = string((*pstate).winnings,   $
   format = format)
win_id = widget_info(event.top,find_by_uname = 'ROLL_WINNINGS')
widget_control,win_id,set_value = string((*pstate).winnings,   $
   format = format)
```

```
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xcac_roll,event
widget_control,event.top,get_uvalue = pstate
names = (*pstate).names
for i = 0,n_elements(names)-1 do begin
   dice_id = widget_info(event.top,find_by_uname = names[i])
   widget_control,dice_id,set_value = -1
   widget_control,dice_id,get_value = dice_value
   if i eq 0 then dice = dice_value else dice = [dice,dice_value]
endfor
bet_id = widget_info(event.top,find_by_uname = 'BET_AMOUNT')
widget_control,bet_id,get_value = bet & bet_amt = fix(bet[0])
dice_number_id = widget_info(event.top,   $
   find_by_uname = 'DICE_NUMBER')
widget_control,dice_number_id,get_value = val
guess = fix(val[0])
m_ind = where(dice eq guess,count)
case count of
0: begin
      winnings = -bet_amt
      (*pstate).purse = (*pstate).purse-bet_amt
   end
1: begin
      winnings = bet_amt
      (*pstate).purse = (*pstate).purse+winnings
   end
2: begin
      winnings = (*pstate).dbl*bet_amt
      (*pstate).purse = (*pstate).purse+winnings
   end
3: begin
      winnings = (*pstate).trpl*bet_amt
      (*pstate).purse = (*pstate).purse+winnings
   end
endcase
(*pstate).winnings = (*pstate).winnings+winnings
format = '(f8.2)'
purse_id = widget_info(event.top,find_by_uname = 'PURSE')
widget_control,purse_id,set_value = string((*pstate).purse,    $
   format = format)
win_id = widget_info(event.top,find_by_uname = 'WINNINGS')
widget_control,win_id,set_value = string((*pstate).winnings,   $
   format = format)
win_id = widget_info(event.top,find_by_uname = 'ROLL_WINNINGS')
widget_control,win_id,set_value = string(winnings,format = format)
if (*pstate).purse le 0.0 then begin
   strout = 'Your purse is empty!  Would you like to restart?'
   ok = dialog_message(dialog_parent = event.top,strout,/question)
   if strupcase(ok) eq 'YES' then begin
      ret = xcac_reset(event)
   endif else begin
      widget_control,event.top,/destroy
   endelse
endif
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xcac_event,event
uname = widget_info(event.id,/uname)
case uname of
'QUIT':  widget_control,event.top,/destroy
'ROLL':  xcac_roll,event
```

```
'RESET': ret = xcac_reset(event)
else:
endcase
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xchuck_a_luck
tlb = widget_base(/tlb_frame_attr,title = 'Chuck-a-luck',/col)
dbl = 2 & trpl = 10
this_font = 'Helvetica*Bold'
strout = 'Payout: Double='+strtrim(string(dbl),2)+ $
   ':1, Triple=' + strtrim(string(trpl),2)+':1'
void = widget_label(tlb,value = strout,font = this_font)
row_base = widget_base(tlb,/row,/align_center)
names = ['DICE_1','DICE_2','DICE_3']
for i = 0,n_elements(names)-1 do $
   void = rmd_dice(row_base,uname = names[i],tumble_cnt = 10, $
      /align_center,/no_press)
void = widget_button(tlb,value = 'Roll Dice',   $
   font = this_font,uname = 'ROLL')
next_row = widget_base(tlb,/row,/align_center)

void = widget_button(next_row,value = 'Reset',  $
   font = this_font,uname = 'RESET')
void = widget_button(next_row,value = 'Quit',   $
   font = this_font,uname = 'QUIT')
col_base = widget_base(tlb,/col,/base_align_right)
void = cw_field(col_base,title = 'Purse ($)',value = 100, $
   uname = 'PURSE',/noedit,font = this_font,fieldfont = this_font)
void = cw_field(col_base,title = 'Bet amount ($)',value = 1,   $
   uname = 'BET_AMOUNT',font = this_font,fieldfont = this_font)
void = cw_field(col_base,title = 'Dice number (1-6)',value = 1, $
   uname = 'DICE_NUMBER',font = this_font,fieldfont = this_font)
void = cw_field(col_base,title = 'Winnings this roll ($)',value = 0, $
   uname = 'ROLL_WINNINGS',/noedit,font = this_font,  $
      fieldfont = this_font)
void = cw_field(col_base,title = 'Overall Winnings ($)',value = 0, $
   uname = 'WINNINGS',/noedit,font = this_font,fieldfont = this_font)

centertlb,tlb
widget_control,tlb,/realize
state = {   purse:100.0,         $
            winnings:0.0,        $
            names:names,         $
            dbl:dbl,             $
            trpl:trpl            }
pstate = ptr_new(state,/no_copy)
widget_control,tlb,set_uvalue = pstate

reg_name = 'xcac'
xmanager,reg_name,tlb,/no_block,event_handler = 'xcac_event', $
   cleanup = 'xcac_cleanup'
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```