Randy Hyde's Win32 Assembly Language Tutorials (Featuring HOWL)

#7: Graphic Objects

In this seventh tutorial of this series, we'll take a look at implementing various graphic objects (such as rectangles, round rectangles, ellipses, pie chart wedges, and polygons, on HOWL forms.

Prerequisites:

This tutorial set assumes that the reader is already familiar with assembly language programming and HLA programming in particular. If you are unfamiliar with assembly language programming or the High Level Assembler (HLA), you will want to grab a copy of my book "The Art of Assembly Language, 2nd Edition" from No Starch Press (www.nostarch.com). The HOWL (HLA Object Windows Library) also makes heavy use of HLA's object-oriented programming facilities. If you are unfamiliar with object-oriented programming in assembly language, you will want to check out the appropriate chapters in "The Art of Assembly Language" and in the HLA Reference Manual. Finally, HOWL is documented in the HLA Standard Library Reference Manual; you'll definitely want to have a copy of the chapter on HOWL available when working through this tutorial.

Source Code:

The source code for the examples appearing in this tutorial are available as part of the HLA Examples download. You'll find the sample code in the Win32/HOWL subdirectories in the unpacked examples download. This particular tutorial uses the files *015_graphics1.hla, 016_graphics2.hla, 017_graphics3.hla, 018_graphics4.hla,* and *019_graphics5.hla*. Though this particular document does not describe *015x_graphics1.hla, 016x_graphics2.hla, 017x_graphics3.hla, 018x_graphics4.hla,* and *019x_graphics5.hla,* you may also find these files of interest when reading through this tutorial.

Graphic Objects:

The HOWL framework supports several objects that draw graphic objects on a form. These graphic objects are generally static objects, though they are subclassed from the `wClickable_t` class, so they support clicking events (but not double-clicking). Generally, an application's designer uses graphic objects to spruce up the look of a form, they aren't typically active elements on the form.

The `wRectangle` type is the simplist graphic object avaiable in the HOWL Declarative Language (HDL). Here is the syntax of the `wRectangle` statement in the HDL:

```
wRectangle
(
   rectName,        // HLA identifier
   x,               // x
   y,               // y
   w,               // width
   h,               // height
```

```
    lineColor,       // linecolor (RGB)
    fillColor        // Rectangle interior color (RGB)
)
```

rectName: this is the identifier name that HOWL uses in the `wForm` declaration for the rectangle object.

x, y, w, and **h:** `these` fields specify the bounding box surrounding the rectangle.

lineColor: this argument specifies an RGB value that HOWL uses when drawing the rectangles's outline.

fillColor: this argument specifies an RGB value that fills the interior of the rectangle.

Rectangle objects in HOWL are unique amongst the graphic objects insofar as they don't support a background color argument. The background color would normally be applied to that area of the bounding box not covered by the graphic object itself. However, rectangles completely cover the bounding box, so there is no background area showing through and therefore no need for a background color.

A round rectangle is a rectangle with rounded corners. Here is the declaration syntax for the `wRoundRect` statement in the HDL:

```
wRoundRect
(
    rrectName,       // HLA identifier
    x,               // x
    y,               // y
    w,               // width
    h,               // height
    cw,              // Corner width
    cht,             // Corner height
    lineColor,       // linecolor (RGB)
    fillColor,       // Round rectangle interior color (RGB)
    BkgColor         // Round rectangle exterior color (RGB)
)
```

rrectName: this argument is the identifier that HOWL uses for the round rectangle object.

x, y, w, and **h:** these fields specify the bounding box surrounding the round rectangle.

cw and **ch:t** these arguments specify the width and height of the ellipse used to draw the corners.

lineColor: the RGB color that HOWL uses to draw the outline of the round rectangle.

fillColor: this argument specifies an RGB value that HOWL uses to fill the interior of the rectangle.

bkgColor: this argument specifies an RGB value that HOWL uses to fill the retangular area described by `x, y, w,` and `h` that is outside the round rectangle. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the round rectangle. If the H.O. byte of this color value is $FF (it is normally zero for RGB values), then HOWL will only paint the background of the round rectangle object when you create the object or when you use the `w.InvalidateRect` Win32 API call (specifying true as the last parameter). This allows you to overlay round rectangle objects over other graphic objects on the form. Note that this transparency feature is not completely automatic. Certain Windows API calls may require your application to redraw the top object in a pair of overlaid graphic object under certain conditions.

You can draw ellipses and circles on a form by using the `wEllipse` HDL statement. Here's the syntax for that call:

```
wEllipse
(
   ellipseName,     // HLA identifier
   x,               // x
   y,               // y
   w,               // width
   h,               // height
   lineColor,       // linecolor (RGB)
   fillColor,       // Ellipse interior color (RGB)
   bkgColor         // Ellipse exterior color (RGB)
)
```

`ellipseName`: this is the identifier name that HOWL uses in the `wForm` declaration for the ellipse object.

`x, y, w,` and **`h:`** these fields specify the bounding box surrounding the ellipse. If this bounding box is a square, then you'll draw a circle on the form.

`lineColor:` specifies an RGB value that HOWL uses when drawing the outline of the ellipse.

`fillColor:` specifies an RGB value that HOWL uses to fill the interior of the ellipse.

`bkgColor:` specifies an RGB value that HOWL uses to fill the bounding box area that is outside the ellipse. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the ellipse. If the H.O. byte of this color value is $FF (it is normally zero for RGB values), then HOWL will only paint the background of the ellipse object when you create the object or when you use the `w.InvalidateRect` Win32 API call (specifying true as the last parameter). This allows you to overlay ellipse objects over other graphic objects on the form. Note that this transparency feature is not completely automatic. Certain Windows API calls may require your application to redraw the top object in a pair of overlaid graphic object under certain conditions.

```
wPie
(
   pieName,         // HLA identifier
   x,               // x
   y,               // y
   w,               // width
   h,               // height
   startAngle,      // Starting handle (in degrees)
   endAngle,        // Ending angle (in degrees)
   lineColor,       // linecolor (RGB)
   fillColor,       // Ellipse interior color (RGB)
   bkgColor         // Ellipse exterior color (RGB)
)
```

`pieName:` this is the identifier name that HOWL uses in the `wForm` declaration for the pie slice object.

`x, y, w,` and **`h:`** these fields specify the bounding box surrounding the pie wedge.

`startAngle:` this parameter is a real64 value that specifies the starting angle of the pie slice. The angle is specified in degrees. Angles are measured in a counter-clockwise fashion from

the vertical line going from the middle of the bounding box to the top of the bounding box (warning: this is not intuitive).

endAngle: this parameter is a real64 value that specifies the ending angle of the pie wedge. The angle is specified in degrees. The `wPie` object draws a slice of a pie graph filling in the ellipse from the `startAngle` to the `endAngle` in a counter-clockwise fashion.

lineColor: specifies an RGB value that HOWL uses when drawing the outline of the pie slice.

fillColor: specifies an RGB value that HOWL uses to fill the interior of the pie slice.

bkgColor: specifies an RGB value that HOWL uses to fill the bounding box that is outside the pie slice. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the pie slice. If the H.O. byte of this color value is $FF (it is normally zero for RGB values), then HOWL will only paint the background of the pie wedge object when you create the object or when you use the `w.InvalidateRect` Win32 API call (specifying true as the last parameter). This allows you to overlay pie wedges objects over other graphic objects on the form. Note that this transparency feature is not completely automatic. Certain Windows API calls may require your application to redraw the top object in a pair of overlaid graphic object under certain conditions.

```
wPolygon
(
   polyName,        // HLA identifier
   x,               // x
   y,               // y
   w,               // width
   h,               // height
   lineColor,       // linecolor (RGB)
   fillColor,       // Ellipse interior color (RGB)
   bkgColor,        // Ellipse exterior color (RGB)
   x1,              // Optional points list
   y1,              // Must have an even number of coordinates
   x2,
   y2,
   .
   .
   .
   xn,
   yn
)
```

polyName: this is the identifier name that HOWL uses in the `wForm` declaration for the polygon object.

x, y, w, and **h:** these fields specify the bounding box surrounding the polygon.

lineColor: specifies an RGB value that HOWL uses when drawing the outline of the polygon.

fillColor: specifies an RGB value that HOWL uses to fill the interior of the polygon.

bkgColor: specifies an RGB value that HOWL uses to fill the part of the bounding box that is outside the polygon. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the polygon. If the H.O. byte of this color value is $FF (it is normally zero for RGB values), then HOWL will only paint the background of the polgon object when you create the object or when you use the `w.Invali-`

`dateRect` Win32 API call (specifying true as the last parameter). This allows you to overlay polygon objects over other graphic objects on the form. Note that this transparency feature is not completely automatic. Certain Windows API calls may require your application to redraw the top object in a pair of overlaid graphic object under certain conditions.

The remain arguments always appear in pairs and specify the points that make up the polygon. If you specify n points (n*2 arguments), HOWL will draw n lines between each pair of points (and between (xn,yn) and (x1,y1) to complete the closed polygon).

Polygons are somewhat special amongst the graphic objects. In the Windows GDI functions, this size of most objects is defined by a bounding box. Change the bounding box parameters and you change the size of the object. For example, if you make the bounding box for a rectangle wider, that rectangle is drawn on the screen to fill in the bounding box. Ditto for ellipses, round rectangles, and pie wedges. For polygons, however, all the bounding box does is affect the clipping region for the polygon. If you shrink the size of the bounding box (such that some line segments of the polygon fall outside the bounding box), then Windows will not shrink the size of the polygon, it will simply clip (not draw) those line segments that fall outside the bounding box.

This inconsistent behavior is unfortunate. A generalized set of graphic primitives behaves the same way for all objects. Under HOWL, the `wPolygon` object is enhanced so that changing the bounding box size will scale the polygon in an appropriate fashion. If your initial set of verticies (points) would create line segments outside the bounding box, Windows will clip those. However, if you change the size of the bounding box after creating a polygon, then HOWL will automatically scale the verticies to fill the new bounding box in a proportional manner.

A Rectangle Example

As noted earlier, the `wRectangle` object is the simplist graphic object you can draw on a form. Therefore, it makes sense to kick off this tutorial with an example of a rectangle object on a form.

The *015_graphics1.hla* source file follows the same pattern as the previous examples in this tutorial. It draws a rectangle and then gives you the ability to show/hide, move, resize, and change the color of the rectangle. Because graphic objects are clickable, the graphic demos (including the rectangle demo) also install an on-click event handler that will change the color of the object when you click on it.

With the preliminary discussion out of the way, let’s look at the HDL code for this example:

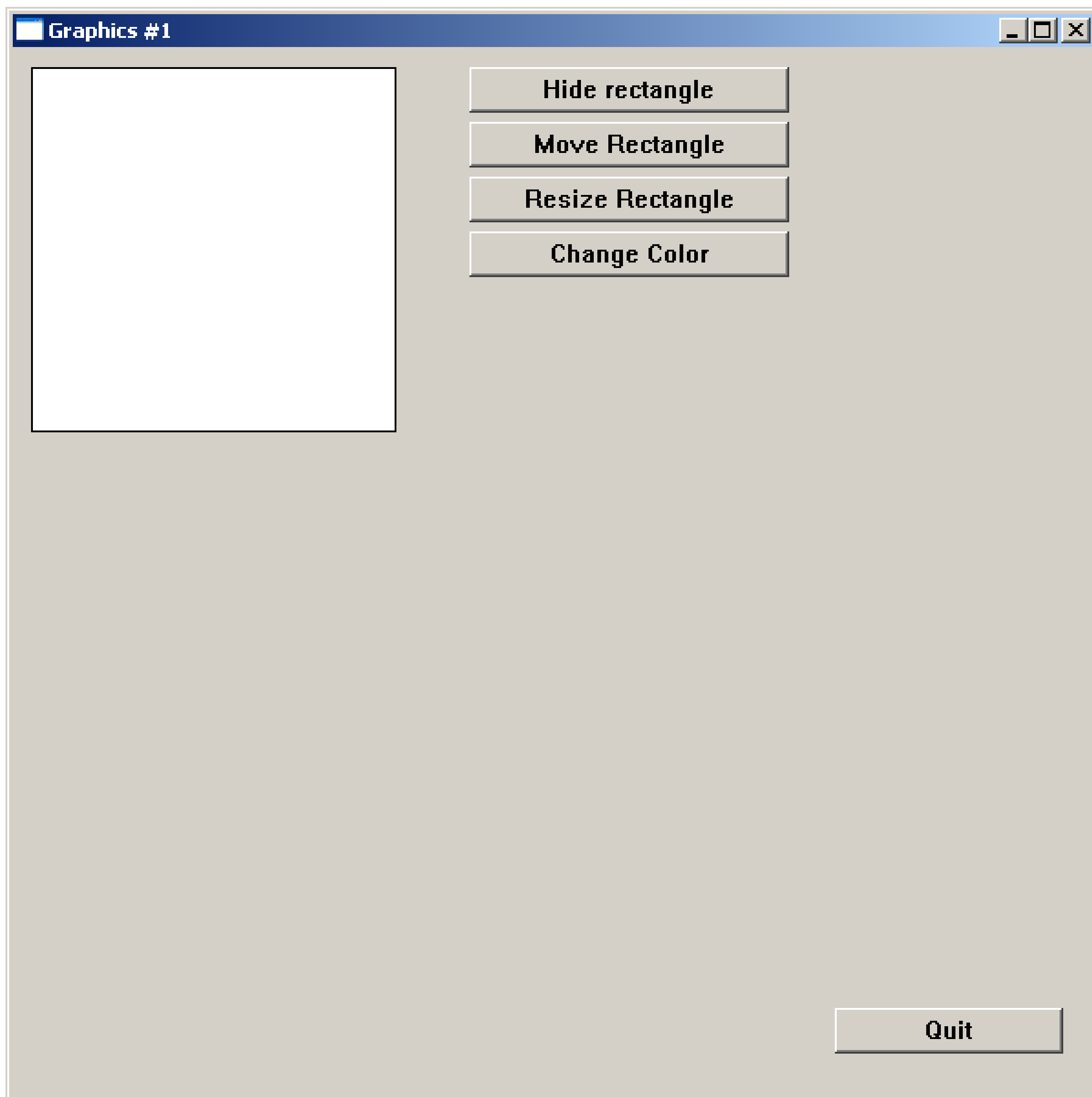
```
wForm( mainAppWindow );

    var
        showState   :boolean;
        align(4);


    wRectangle
    (
        rectangle1,
        10,
        10,
        200,
        200,
        RGB( 0, 0, 0 ),
```

```
    RGB( 255, 255, 255 )
)

wPushButton
(
    button2,                    // Field name in mainWindow object
    "Hide rectangle",           // Caption for push button
    250,                        // x position
    10,                         // y position
    175,                        // width
    25,                         // height
    hideShowRectangle               // initial "on click" event handler
)


wPushButton
(
    button3,                    // Field name in mainWindow object
    "Move Rectangle",               // Caption for push button
    250,                        // x position
    40,                         // y position
    175,                        // width
    25,                         // height
    moveRectangle                   // initial "on click" event handler
)


wPushButton
(
    button4,                    // Field name in mainWindow object
    "Resize Rectangle",             // Caption for push button
    250,                        // x position
    70,                         // y position
    175,                        // width
    25,                         // height
    resizeRectangle                 // initial "on click" event handler
)


wPushButton
(
    button5,                    // Field name in mainWindow object
    "Change Color",             // Caption for push button
    250,                        // x position
    100,                        // y position
    175,                        // width
    25,                         // height
    colorRectangle                  // initial "on click" event handler
)


// Place a quit button in the lower-right-hand corner of the form:

wPushButton
(
    quitButton,             // Field name in mainWindow object
    "Quit",                 // Caption for push button
    450,                    // x position
    525,                    // y position
```

```
        125,                        // width
        25,                         // height
        onQuit                      // "on click" event handler
    )

endwForm
```

About the only thing different in this example from all the previous tutorials is the presence of the `wRectangle` statement. Based on the discussion earlier on the HDL wRectangle statement, it should be pretty easy for you to figure out what this statement is doing on the form.

Here's the display this program generates:



The button press event handlers are all almost identical to their corresponding procedures in the previous tutorial examples in this series, so it's not worth any space to further describe them at this poing. Two procedures, however, are worth taking a look at: the onCreate method and the onClick widgetProc:

```
method mainAppWindow_t.onCreate;
begin onCreate;

    // Initialize the showState data field:

    mov( false, this.showState );

    // Install onClick handler for the graphic object:

    mov( (type mainAppWindow_t [esi]).rectangle1, esi );
    (type wRectangle_t [esi]).set_onClick( &onClick );


end onCreate;
```

The interesting thing to note about the `onCreate` method is that it calls `set_onClick` for the `wRectangle_t` object in order to install an on-click event handler for the rectangle. This is necessary because the `wRectangle` HDL statement initializes the on-click handler to NULL and doesn't provide a way for you to set this handler without making an explicit `set_onClick` method call.

The `onClick` widgetProc (referenced in the code above) takes the following form:

```
proc onClick:widgetProc;
begin onClick;

    stdout.put( "Clicked on graphic object" nl );
    mov( thisPtr, esi );
    (type wRectangle_t [esi]).set_lineColor( RGB( 0, 0, 255 ));
    (type wRectangle_t [esi]).set_fillColor( RGB( 255, 0, 255 ));

end onClick;
```

Note that this code sets the rectangle's fill color to magenta (red+blue) and the line color to blue. Clicking on the object does not toggle between any colors, it just sets it to magenta and blue (the user can change the color from magenta and blue by pressing the "change color" button). Here is the complete code for the *015_graphics1.hla* program:

```
// graphics1-
//
//  This program demonstrates the use of wRectangles on a form.

program graphics1;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

?@NoDisplay     := true;
?@NoStackAlign  := true;

#includeOnce( "stdlib.hhf" )
#includeOnce( "howl.hhf" )
```

```
const
    applicationName := "Graphics #1";
    formX           := w.CW_USEDEFAULT; // Let Windows position this guy
    formY           := w.CW_USEDEFAULT;
    formW           := 600;
    formH           := 600;




// Forward declarations for the onClick widgetProcs that we're going to
// call when an event occurs.

proc hideShowRectangle      :widgetProc; @forward;
proc moveRectangle          :widgetProc; @forward;
proc resizeRectangle        :widgetProc; @forward;
proc colorRectangle         :widgetProc; @forward;
proc onClick                :widgetProc; @forward;
proc onQuit                 :widgetProc; @forward;



// Here's the main form definition for the app:

wForm( mainAppWindow );

    var
        showState   :boolean;
        align(4);


    wRectangle
    (
        rectangle1,
        10,
        10,
        200,
        200,
        RGB( 0, 0, 0 ),
        RGB( 255, 255, 255 )
    )

    wPushButton
    (
        button2,                    // Field name in mainWindow object
        "Hide rectangle",           // Caption for push button
        250,                        // x position
        10,                         // y position
        175,                        // width
        25,                         // height
        hideShowRectangle               // initial "on click" event handler
    )


    wPushButton
    (
        button3,                    // Field name in mainWindow object
        "Move Rectangle",               // Caption for push button
```

```
        250,                        // x position
        40,                         // y position
        175,                        // width
        25,                         // height
        moveRectangle                   // initial "on click" event handler
    )


    wPushButton
    (
        button4,                    // Field name in mainWindow object
        "Resize Rectangle",             // Caption for push button
        250,                        // x position
        70,                         // y position
        175,                        // width
        25,                         // height
        resizeRectangle                 // initial "on click" event handler
    )


    wPushButton
    (
        button5,                    // Field name in mainWindow object
        "Change Color",              // Caption for push button
        250,                        // x position
        100,                        // y position
        175,                        // width
        25,                         // height
        colorRectangle                  // initial "on click" event handler
    )




    // Place a quit button in the lower-right-hand corner of the form:

    wPushButton
    (
        quitButton,             // Field name in mainWindow object
        "Quit",                 // Caption for push button
        450,                    // x position
        525,                    // y position
        125,                    // width
        25,                     // height
        onQuit                  // "on click" event handler
    )

endwForm


// Must invoke the following macro to emit the code generated by
// the wForm macro:

mainAppWindow_implementation();
```

```
// The colorRectangle widget proc will change the foreground and background color.

proc colorRectangle:widgetProc;
begin colorRectangle;

    mov( mainAppWindow.rectangle1, esi );
    (type wRectangle_t [esi]).get_lineColor();
    if( eax = RGB( 0, 0, 0 )) then

        (type wRectangle_t [esi]).set_lineColor( RGB( 0, 255, 0 ));
        (type wRectangle_t [esi]).set_fillColor( RGB( 255, 0, 0 ));

    else

        (type wRectangle_t [esi]).set_lineColor( RGB( 0, 0, 0 ));
        (type wRectangle_t [esi]).set_fillColor( RGB( 255, 255, 255 ));

    endif;


end colorRectangle;



// The resizeRectangle widget proc will resize label1 between widths 150 and 200.

proc resizeRectangle:widgetProc;
begin resizeRectangle;

    mov( mainAppWindow.rectangle1, esi );
    (type wLabel_t [esi]).get_width();
    if( eax = 200 ) then

        stdout.put( "Resizing rectangle to width/height 150" nl );
        (type wRectangle_t [esi]).resize( 150, 150 );

    else

        stdout.put( "Resizing label to width/height 200" nl );
        (type wRectangle_t [esi]).resize( 200, 200 );

    endif;

end resizeRectangle;



// The moveRectangle widget proc will move label
// between y positions 10 and 40.

proc moveRectangle:widgetProc;
begin moveRectangle;

    mov( mainAppWindow.rectangle1, esi );
    (type wRectangle_t [esi]).get_y();
    if( eax = 10 ) then
```

```
        stdout.put( "Moving rectangle to y-position 40" nl );
        (type wRectangle_t [esi]).set_y( 40 );

    else

        stdout.put( "Moving rectangle to y-position 10" nl );
        (type wRectangle_t [esi]).set_y( 10 );

    endif;

end moveRectangle;



// The hideShowRectangle widget proc will hide and show rectangle1.

proc hideShowRectangle:widgetProc;
begin hideShowRectangle;

    mov( thisPtr, esi );
    if( mainAppWindow.showState ) then

        (type wPushButton_t [esi]).set_text( "Hide Rectangle" );
        mov( false, mainAppWindow.showState );
        stdout.put( "Showing rectangle 1" nl );

        mov( mainAppWindow.rectangle1, esi );
        (type wRectangle_t [esi]).show();

    else

        (type wPushButton_t [esi]).set_text( "Show Rectangle" );
        mov( true, mainAppWindow.showState );
        stdout.put( "Hiding rectangle 1" nl );

        mov( mainAppWindow.rectangle1, esi );
        (type wRectangle_t [esi]).hide();

    endif;

end hideShowRectangle;





// Here's the onClick event handler the graphic object:

proc onClick:widgetProc;
begin onClick;

    stdout.put( "Clicked on graphic object" nl );
    mov( thisPtr, esi );
    (type wRectangle_t [esi]).set_lineColor( RGB( 0, 0, 255 ));
    (type wRectangle_t [esi]).set_fillColor( RGB( 255, 0, 255 ));

end onClick;
```

```
// Here's the onClick event handler for our quit button on the form.
// This handler will simply quit the application:

proc onQuit:widgetProc;
begin onQuit;

    // Quit the app:

    w.PostQuitMessage( 0 );

end onQuit;





// We'll use the main application form's onCreate method to initialize
// the various buttons on the form.
//
// This could be done in appStart, but better to leave appStart mainly
// as boilerplate code. Also, putting this code here allows us to use
// "this" to access the mainAppWindow fields (a minor convenience).

method mainAppWindow_t.onCreate;
begin onCreate;

    // Initialize the showState data field:

    mov( false, this.showState );

    // Install onClick handler for the graphic object:

    mov( (type mainAppWindow_t [esi]).rectangle1, esi );
    (type wRectangle_t [esi]).set_onClick( &onClick );


end onCreate;



///////////////////////////////////////////////////////////////////////////////
//
//
// The following is mostly boilerplate code for all apps (about the only thing
// you would change is the size of the main app's form)
//
//
///////////////////////////////////////////////////////////////////////////////
//
// When the main application window closes, we need to terminate the
// application. This overridden method handles that situation.  Notice the
// override declaration for onClose in the wForm declaration given earlier.
// Without that, mainAppWindow_t would default to using the wVisual_t.onClose
// method (which does nothing).

method mainAppWindow_t.onClose;
```

```
begin onClose;

    // Tell the winmain main program that it's time to terminate.
    // Note that this message will (ultimately) cause the appTerminate
    // procedure to be called.

    w.PostQuitMessage( 0 );


end onClose;






// When the application begins execution, the following procedure
// is called.  This procedure must create the main
// application window in order to kick off the execution of the
// GUI application:

procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

    mainAppWindow.create_mainAppWindow
    (
        applicationName,        // Window title
        w.WS_EX_CONTROLPARENT,  // Need this to support TAB control selection
        w.WS_OVERLAPPEDWINDOW,  // Style
        NULL,                   // No parent window
        formX,                  // x-coordinate for window.
        formY,                  // y-coordinate for window.
        formW,                  // Width
        formH,                  // Height
        howl.bkgColor_g,        // Background color
        true                    // Make visible on creation
    );
    mov( esi, pmainAppWindow ); // Save pointer to main window object.
    pop( esi );

end appStart;



// appTerminate-
//
//  Called when the application is quitting, giving the app a chance
// to clean up after itself.
//
// Note that this is called *after* the mainAppWindow_t.onClose method
// executes (indeed, mainAppWindow_t.onClose, by posting the quit message,
// is what actually causes the program to begin terminating, which leads
// to the execution of this procedure).

procedure appTerminate;
begin appTerminate;
```

```
    // Clean up the main application's form.
    // Note that this will recursively clean up all the widgets on the form.

    mainAppWindow.destroy();

end appTerminate;


// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;



// The main program for a HOWL application must
// call the HowlMainApp procedure.

begin graphics1;

    // Set up the background and transparent colors that the
    // form will use when registering the window_t class:

    w.GetSysColor( w.COLOR_MENU );
    mov( eax, howl.bkgColor_g );
    or( $FF00_0000, eax );
    mov( eax, howl.transparent_g );
    w.CreateSolidBrush( howl.bkgColor_g );
    mov( eax, howl.bkgBrush_g );

    // Start the HOWL Framework Main Program:

    HowlMainApp();

    // Delete the brush we created earlier:

    w.DeleteObject( howl.bkgBrush_g );

end graphics1;
```

The HOWL wSurface_t, wFilledFrame_t, wEllipse_t, wPie_t, wPolygon_t, wRectangle_t, and wRoundRect_t Classes

Okay, let's spend a few moments looking at the classes responsible for the graphic objects on a HOWL form. The first class to consider is the `wSurface_t` abstract base class:

```
wSurface_t:
    class inherits( wClickable_t );

        var
            align( 4 );
            wSurface_private:
                record

                    // onPaint event pointer:

                    onPaint     :widgetProc;

                endrecord;

        procedure create_wSurface
        (
            wsName      :string;
            exStyle     :dword;
            style       :dword;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            bkgColor    :dword;
            visible     :boolean
        );  external;

        override method destroy;                        external;
        override method processMessage;                 external;
        override method onClose;                        external;
        override method onCreate;                       external;

        method get_onPaint;     @returns( "eax" );      external;
        method set_onPaint( onPaint:widgetProc );       external;

    endclass;
```

The first thing to note about the `wSurface_t` class is that it inherits the fields from `wClickable_t`. We've already looked at the `wClickable_t` class in previous tutorials in this series (all the button classes are subclasses of `wClickable_t`), so we won't consider that class here.

The `wSurface_t` class as a single private data field: `onPaint`. This is a pointer to a widgetProc that `wSurface_t` objects can call (assuming the pointer contains a non-NULL value) when Windows wants to draw a `wSurface_t` object. For most graphic objects this data field will always contain NULL because the `processMessage` method of the particular classes is going to handle drawing the graphic objects for you. However, the onPaint pointer provides a convenient way for you to override the drawing process on an object-by-object basis. Note that if this field contains a non-NULL value, HOWL will call the `onPaint` widgetProc instead of calling the graphic object's processMessage method (which normally handles drawing the object).

A `wSurface_t` object is basically a "window" in Microsoft Windows. It is essentially the same thing as a `window_t` object except that it is not derived from `wContainer_t` and (therefore), it cannot contain other widgets. `wSurface_t` subclasses are generally expected to draw something on the `wSurface_t` window using Win32 GDI API calls. A discussion of the

Win32 GDI API calls is beyond the scope of this tutorial, for more information on Windows GDI programming, check out "Windows Programming In Assembly Language" on Webster (http://webster.cs.ucr.edu, follow the link on "Windows Programming." It's the same page where this tutorial is based. A class derived from wSurface_t will process w.WM_PAINT messages to draw the object (a call to the onPaint widgetProc occurs when a w.WM_PAINT message comes a long), so all the rules associated with handling w.WM_PAINT messages apply in those methods/widgetProcs. Again, see "Windows Programming in Assembly" or documentation on the Win32 GDI API for more details.

Because `wSurface_t` is an abstract base class, application programs are unlikely to call the class constructor (unless those applications are creating user-defined derived classes from `wSurface_t`). While it's not all that rare for applications to create their own subclasses of `wSurface_t` (or `wFilledFrame_t`, the next class we're going to look at), doing so is beyond the scope of this tutorial so I'll skip over an in-depth look at the constructor function.

As usual, the destroy and processMessage methods are of little interest to an application programmer. These methods are called automatically by HOWL in response to other events. Likewise, an application will probably never call the `onCreate` or `onClose` methods (they're empty). These exist so that someone creating their own subclass can override them and provide some functionality when the class is created or destroyed.

The `get_onPaint` and `set_onPaint` methods are of interest to application programmers. These are the accessor and mutator functions for the `onPaint` private data fields. An application will call these functions if it wishes to access that data field.

The main thing that the `wSurface_t` class provides is a window on which to draw things. The next class we're going to look at, `wFilledFrame_t`, refines this definition a bit to describe objects that are closed shapes with an outline and a fill color. Here's the class definition for the `wFilledFrame_t` type:

```
wFilledFrame_t:
    class inherits( wSurface_t );
        var
            align( 4 );
            wFilledFrame_private:
                record

                    lineColor   :dword;
                    fillColor   :dword;

                    _linePen    :dword;
                    _lineBrush  :dword;
                    _fillBrush  :dword;

                endrecord;


        procedure create_wFilledFrame
        (
            wrName      :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            lineColor   :dword;
            fillColor   :dword;
            bkgColor    :dword
```

```
        );  external;

        method get_fillColor;   @returns( "eax" );      external;
        method get_lineColor;   @returns( "eax" );      external;

        method set_fillColor( fillColor:dword );        external;
        method set_lineColor( lineColor:dword );        external;

        override method destroy;                        external;
        override method processMessage;                 external;

    endclass;
```

The `wFilledFrame_t` class is another abstract base class. As noted above, it describes closed objects that have an outline and a fill color (hence the name `wFilledFrame_t`). The `lineColor` and `fillColor` private data fields contain the RGB colors that HOWL will use to draw the outline of the object (`lineColor`) and fill the object (`fillColor`). The `get_fillColor` and `get_lineColor` methods are the accessor functions and the `set_fillColor` and `set_lineColor` methods are the mutators for this private data field values.

The remaining three data fields, `_linePen`, `_lineBrush`, and `_fillBrush` are private data fields computed from the fill and line colors. Note that you should always call the accessor and mutator functions when changing the line and fill colors so that HOWL can keep these pen and brush values consistent.

As for `wSurface_t`, we're not going to discuss the constructor here because application programs should never be calling it. The parameters are pretty obvious, but there really isn't much you can do with a `wFilledFrame_t` object unless you subclass it and provide a processMessage method to actually draw something in its window.

Before going any farther, it's worth pointing out one important feature concerning the background color argument to the `wFilledFrame_t` constructor. Most of the standard HOWL graphics objects (`wRectangle_t` being the exception) allow a very special form of a background color. Generally, a background color is an RGB value consuming the L.O. 24 bits of a dword (with the H.O. 8 bits containing zero). The `wEllipse_t, wPie_t, wPolygon_t,` and `wRoundRect_t` drawing code look for a special value in the H.O. byte of the bkgColor value. If this byte contains $FF, then the drawing code for these objects will only draw the background for the object when you first create it or when you explicitly tell Windows to redraw the background by using a call to `w.InvalidateRect( objectHandle, NULL, true);` If the H.O. byte contains zero (the only other legal value), then the drawing code will erase the bounding box area before redrawing the object. By placing $FF in the H.O. byte of the background color, you get a "transparent" background in certain cases. You'll see that this isn't a perfect implementation of a transparent background color, but it does give you the ability to overlay various graphic object and not have their bounding boxes interfere with one another. You'll see some examples of this feature in the remaining examples in this tutorial.

The next class to consider in our journey is the `wEllipse_t` class. Here's its definition:

```
wEllipse_t:
    class inherits( wFilledFrame_t );

        procedure create_wEllipse
        (
            wrName      :string;
            parent      :dword;
            x           :dword;
```

```
            y           :dword;
            width       :dword;
            height      :dword;
            lineColor   :dword;
            fillColor   :dword;
            bkgColor    :dword
        );  external;

        override method processMessage;                 external;

    endclass;
```

This is a very simple and straight-forward class definition; indeed, this is typical of all the graphic object classes we're going to look at. Most of the class' data and functionality is inherited from `wFilledFrame_t`; each graphic object is going to supply a constructor, it's going to overload the `processMessage` method (which is responsible for actually drawing the object), and in the case of some objects, we'll need some additional data fields to represent that object (ellipses need no such extra data).

The constructor, `wEllipse.create_wEllipse` as the following arguments:

wrName: this is a string holding the HLA identifier of the object. The constructor will initialize the `_name` field with this value.
parent: this is the handle of the window that will hold the `wEllipse` object. This is typically the main form or some other `wContainer_t` object.
x, y, width, and **height**: these arguments specify the bounding box surrounding the ellipse.
lineColor: this is an RGB value that specifies the color of the ellipse's outline.
fillColor: this is an RGB value that specifies the color of the ellipse's interior.
bkgColor: this is an RGB value that specifies the color of the bounding box area that is outside the ellipse. If the H.O. byte of this argument contains $FF (rather than zero), then the `processMessage` method will only paint the background when the object is first created or when you explicitly tell Windows to paint it via an `w.InvalidateRect` call (with true as the last argument).

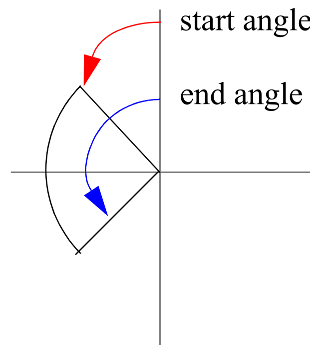
The `processMessage` method (which applications never call) handles the w.WM_PAINT message that is responsible for actually drawing the ellipse on the form.

The `wPie_t` class lets you create pie graph wedges. Here is its class definition:

```
wPie_t:
    class inherits( wFilledFrame_t );

        var
            align( 8 );
            wPie_private:
                record

                    startAngle  :real64;
                    endAngle    :real64;

                endrecord;

        procedure create_wPie
        (
```

```
        wrName      :string;
        parent      :dword;
        x           :dword;
        y           :dword;
        width       :dword;
        height      :dword;
        startAngle  :real64;
        endAngle    :real64;
        lineColor   :dword;
        fillColor   :dword;
        bkgColor    :dword
    );  external;


    override method processMessage;                 external;

    method get_startAngle;  @returns( "st0" );      external;
    method get_endAngle;    @returns( "st0" );      external;

    method set_startAngle( startAngle:real64 );     external;
    method set_endAngle( endAngle:real64 );         external;

endclass;
```

The `startAngle` and `endAngle` private data fields specify the starting and ending angles that define the pie slice. These angles are measured from the vertical bar in a counter-clockwise direction:



The `wPie_t.create_wPie` constructor has the same parameters as the ellipse constructor with the addition of the `startAngle` and `endAngle` arguments. Note that you must specify these angle values in degrees, not radians.

The `processMessage` method, which applications should never directly call, is responsible for processing the `w.WM_PAINT` message that tells the object to actually draw the pie wedge.

The `get_startAngle` and `get_endAngle` methods are the accessor functions for the `startAngle` and `endAngle` private data fields. Note that these accessor functions return their `real64` values on the top of the FPU stack, not in the EAX register (where most accessor functions return their result).

The set_startAngle and set_endAngle methods are the mutator functions that let you set the value of the starting and ending angles for the pie wedge. The angular arguments passed to these methods must be specified in degrees.

You should always call the mutator and accessor methods when accessing the private `startAngle` and `endAngle` values. Internally, HOWL maintains these angles in radians and the accessor/mutator functions handle the conversion between degrees and radians (as well as redrawing the pie wedge whenever you change it).

The `wPolygon_t` class has the following type definition:

```
ptArray :pointer to w.POINT;

wPolygon_t:
    class inherits( wFilledFrame_t );

        var
            align( 4 );
            wPolygon_private:
                record

                    points          :ptArray;
                    scaledPoints    :ptArray;
                    nPoints         :uns32;
                    origW           :dword;
                    origH           :dword;

                endrecord;

        procedure create_wPolygon
        (
            wrName      :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            lineColor   :dword;
            fillColor   :dword;
            bkgColor    :dword
        );  external;


        override method destroy;                         external;
        override method processMessage;                  external;

        override method set_width;                       external;
        override method set_height;                      external;
        override method resize;                          external;

        method set_points
        (
            nPoints :dword;
            points  :ptArray
        );  external;

        method get_points;      @returns( "eax" );      external;
        method get_nPoints;     @returns( "eax" );      external;


    endclass;
```

The points and scaledPoints data fields are pointers to an array of w.POINT values. These arrays contain nPoints elements each. The points array contains the original list of points that the user provides that defines the verticies of the polyon. If the user ever resizes the polygon, then scaledPoints will contain the verticies that HOWL will actually draw, representing the transformed verticies based on the new size. HOWL keeps the scaled vertex values separate from the original points to avoid truncation errors. Whenever the user resizes (scales) the polygon, the scaledPoints array values are always computed from the original points array rather than scaling the previously scaled points. The origW and origH fields contain the original width and height of the bounding box. Whenever the program resizes the polygon, the scaling is computed as the ratio of the original width and height to the new width and height; then HOWL applies this scaling factor to the original points to obtain the scaled points.

The create_wPolygon constructor creates a polygon object but unlike the other graphic object constructors, this one doesn't draw the polygon because it doesn't complete the creation of the polygon (technically speaking, it *does* draw the polygon, but the constructor initializes the polygon with zero points so there is nothing to draw). The create_wPolygon constructor has the usual arguments:

wrName: this is a string holding the HLA identifier of the object. The constructor will initialize the _name field with this value.
parent: this is the handle of the window that will hold the wPolygon object. This is typically the main form or some other wContainer_t object.
x, y, width, and **height**: these arguments specify the bounding box surrounding the polygon.
lineColor: this is an RGB value that specifies the color of the polygon's outline.
fillColor: this is an RGB value that specifies the color of the polygon's interior.
bkgColor: this is an RGB value that specifies the color of the bounding box area that is outside the polygon. If the H.O. byte of this argument contains $FF (rather than zero), then the processMessage method will only paint the background when the object is first created or when you explicitly tell Windows to paint it via an w.InvalidateRect call (with true as the last argument).

As usual, the destroy method is the class' destructor. Generally applications will not call this method directly. Instead, the wContainer_t object containing the polygon will take responsibility for calling the destructor when the container object is destroyed.

The processMessage method is a private method that Windows automatically invokes when it wants the application to draw the polygon. Applications must not call this method directly.

The set_height, set_width, and resize (overridden) methods are the mutator functions for the (inherited) width and height private data fields. This methods are overridden because they need to create a new scaledPoints array whenever the caller changes the size of the bounding box.

The get_points method returns the value of the points private data field in the EAX register. Note that this is the pointer to the actual data that wPolygon_t uses and you must treat this as a read-only data object. If you change the values of any data points, you may get inconsistent results.

The get_nPoints method is the accessor function for the nPoints private data field. It returns the number of data points pointed at by points in the EAX register.

The set_points method is the "mutator" for the points and nPoints data fields. This method creates a copy of the array pointed at by the points argument and stores a pointer to

this new array in the `points` private data field; this method also copies the `nPoints` argument to the `nPoints` private data field. Because this method makes a copy of the array of points, any changes the caller makes to its copy of the points data array that it passes as an argument will not be reflected in the internal `wPolygon_t` data structure. The only way to change the internal points array is by calling `set_points`.

The `wRectangle_t` class has the following definition:

```
wRectangle_t:
    class inherits( wFilledFrame_t );

        procedure create_wRectangle
        (
            wrName      :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            lineColor   :dword;
            fillColor   :dword
        );  external;


        override method processMessage;                 external;

    endclass;
```

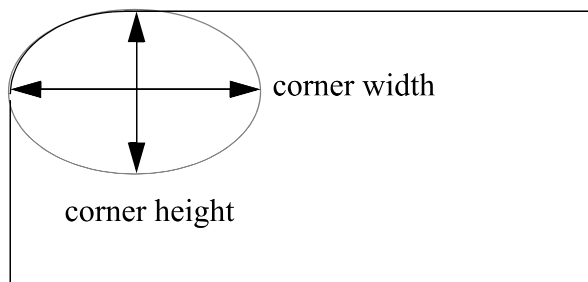
Everything is identical to the `wEllipse_t` class except, of course, that the `processMessage` method will draw a rectangle rather than an ellipse in response to a Windows `w.WM_PAINT` message.

The `wRoundRect_t` class has the following definition:

```
wRoundRect_t:
    class inherits( wFilledFrame_t );

        var
            align( 4 );
            wRoundRect_private:
                record

                    cornerWidth     :dword;
                    cornerHeight    :dword;

                endrecord;

        procedure create_wRoundRect
        (
            wrName          :string;
            parent          :dword;
            x               :dword;
            y               :dword;
            width           :dword;
            height          :dword;
            cornerWidth     :dword;
            cornerHeight    :dword;
```

```
        lineColor       :dword;
        fillColor       :dword;
        bkgColor        :dword
    );  external;

    method get_cornerWidth;      @returns( "eax" );  external;
    method get_cornerHeight;     @returns( "eax" );  external;

    method set_cornerWidth( cornerWidth:dword );     external;
    method set_cornerHeight( cornerHeight:dword );  external;

    override method processMessage;                  external;

endclass;
```

The cornerWidth and cornerHeight private data field variables (and constructor parameters) specify a “mini bounding box” that surrounds the ellipses used to create the rounded corners of the round rectangle:



The constructor for round rectangles is identical to that for the wRectangle_t class with the addition of the cornerWidth and cornerHeight arguments that let you specify the shape of the round rectangle’s corners.

The get_cornerWidth and get_cornerHeight methods are accessor functions that return the values of the cornerWidth and cornerHeight private data fields in the EAX register. The set_cornerWidth and set_cornerHeight methods are the mutator functions that let you set the shape of the round rectangle corners. These mutators will also force a redraw of the round rectangle on the form.

As usual, the processMessage method is a private function that Windows invokes when it wants the round rectangle object to draw a copy of itself on the form. Applications must not directly call this method.

The Graphics2 Application

The *016_graphics2.hla* source file demonstrates a couple of things. First of all, it demonstrates drawing a couple of ellipse graphic objects on a form. Second, it demonstrates using transparent background colors in a graphic object and the steps you have to go through in order to overlay two graphic objects (without the background of one of the objects overwriting the other object).

Structurally, *016_graphics2.hla* is quite similar to *015_graphics1.hla*. The first difference occurs in the HDL code where two `wEllipse` statements replace the single `wRectangle` statement of the previous example in this tutorial:

```
wForm( mainAppWindow );

 var
     showState   :boolean;
     align(4);


 wEllipse
 (
     ellipse1,
     10,
     10,
     200,
     200,
     RGB( 0, 0, 0 ),
     RGB( 255, 255, 255 ),
     howl.transparent_g
 )

 wEllipse
 (
     ellipse2,
     35,
     100,
     200,
     200,
     RGB( 0, 0, 0 ),
     RGB( 255, 255, 255 ),
     howl.transparent_g
 )

 wPushButton
 (
     button2,                    // Field name in mainWindow object
     "Hide ellipse",             // Caption for push button
     250,                        // x position
     10,                         // y position
     175,                        // width
     25,                         // height
     hideShowEllipse             // initial "on click" event handler
 )


 wPushButton
 (
     button3,                    // Field name in mainWindow object
     "Move Ellipse",             // Caption for push button
     250,                        // x position
     40,                         // y position
     175,                        // width
     25,                         // height
     moveEllipse                 // initial "on click" event handler
 )
```

```
    wPushButton
    (
        button4,                    // Field name in mainWindow object
        "Resize Ellipse",               // Caption for push button
        250,                        // x position
        70,                         // y position
        175,                        // width
        25,                         // height
        resizeEllipse                   // initial "on click" event handler
    )


    wPushButton
    (
        button5,                    // Field name in mainWindow object
        "Change Color",             // Caption for push button
        250,                        // x position
        100,                        // y position
        175,                        // width
        25,                         // height
        colorEllipse                    // initial "on click" event handler
    )


    // Place a quit button in the lower-right-hand corner of the form:

    wPushButton
    (
        quitButton,             // Field name in mainWindow object
        "Quit",                 // Caption for push button
        450,                    // x position
        525,                    // y position
        125,                    // width
        25,                     // height
        onQuit                  // "on click" event handler
    )

endwForm
```

In addition to drawing ellipses, you'll notice that these two calls specify `howl.transparent_g` as the background color rather than the usual `howl.bkgColor_g` value. This is the same RGB value as `howl.bkgColor_g`, though the H.O. byte of `howl.transparent_g` contains the value $FF rather than zero. HOWL uses this as a flag to skip drawing the background for a graphic object except in the two special cases of initial object creation and when you explicitly force a redraw of an object with `w.InvalidateRect` and you supply true as the last parameter to `w.InvalidateRect`.

Note that the second ellipse will overlap the first when the two are drawn. This will create some problems when we manipulate `ellipse1` in this application. Changing one of `ellipse1`'s attributes, such as its position, size, or color, will force a redraw of `ellipse1`. Unfortunately, the code that draws `ellipse1` has no clue that `ellipse2` is currently covering up part of `ellipse1`. As such, when the `ellipse1` object redraws itself, it covers up part of `ellipse2` (which is not correct). In order to correct this mistake, any operation that forces a redraw of `ellipse1` must also force a redraw of `ellipse2` (after drawing `ellipse1`) so that `ellipse2` is overlaid on `ellipse1` rather than the other way around. Consider the code for the

colorEllipse widgetProc that changes the color of ellipse1 (and definitely cause ellipse1 to be redrawn):

```
proc colorEllipse:widgetProc;
begin colorEllipse;

    mov( mainAppWindow.ellipse1, esi );
    (type wEllipse_t [esi]).get_lineColor();
    if( eax = RGB( 0, 0, 0 )) then

        (type wEllipse_t [esi]).set_lineColor( RGB( 0, 255, 0 ));
        (type wEllipse_t [esi]).set_fillColor( RGB( 255, 0, 0 ));

    else

        (type wEllipse_t [esi]).set_lineColor( RGB( 0, 0, 0 ));
        (type wEllipse_t [esi]).set_fillColor( RGB( 255, 255, 255 ));

    endif;


    // If we change ellipse1, we've got to force a redraw
    // of ellipse2:

    mov( mainAppWindow.ellipse2, esi );
    w.InvalidateRect( (type wEllipse_t [esi]).handle, NULL, false);

end colorEllipse;
```

Notice the last two statements. The two statements force a redraw of ellipse2 without repainting its background (notice that the last w.InvalidateRect parameer is false). The set_lineColor and set_fillColor methods will cause ellipse1 to be redrawn; the w.InvalidateRect call forces ellipse2 to be redrawn so that it remains on top of ellipse1.

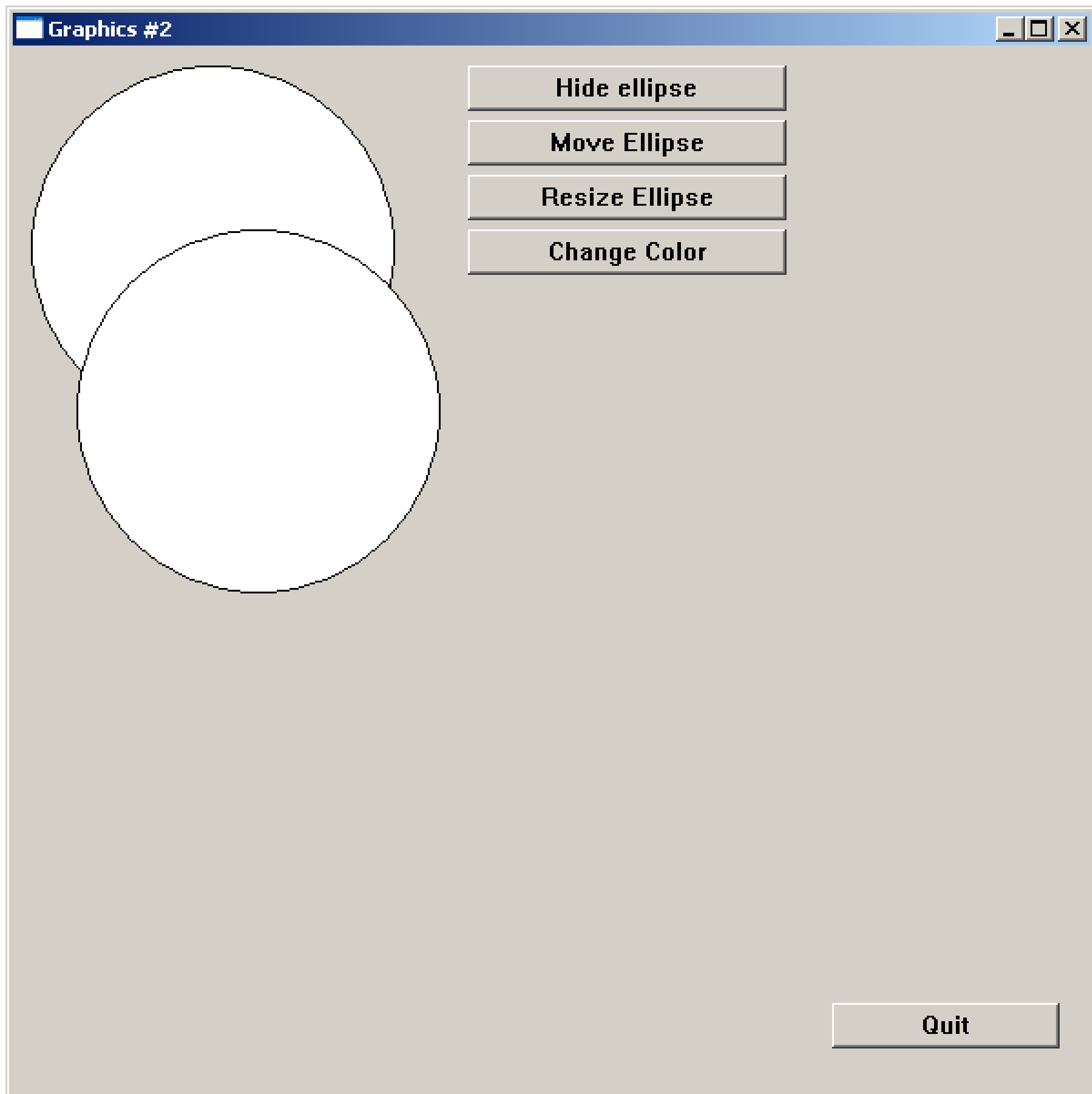
Whenever you move a graphic object the HOWL library code calls the w.MoveWindow API function. Windows will do a “bit-blt” operation to move the window around and this will often drag along artifacts of other objects that overlay the object being moved. Therefore, if you move an object, it’s a wise idea to redraw the object and everything that is on top of it. As an example, here’s the moveEllipse widgetProc:

```
proc moveEllipse:widgetProc;
begin moveEllipse;

    mov( mainAppWindow.ellipse1, esi );
    (type wEllipse_t [esi]).get_y();
    if( eax = 10 ) then

        stdout.put( "Moving ellipse to y-position 40" nl );
        (type wEllipse_t [esi]).set_y( 40 );

    else

        stdout.put( "Moving ellipse to y-position 10" nl );
        (type wEllipse_t [esi]).set_y( 10 );

    endif;
```

```
    // If we move ellipse1, we've got to force a redraw
    // of ellispe1 and ellipse2 in the proper order:

    w.InvalidateRect( (type wEllipse_t [esi]).handle, NULL, true);
    mov( mainAppWindow.ellipse2, esi );
    w.InvalidateRect( (type wEllipse_t [esi]).handle, NULL, false);


end moveEllipse;
```

Notice that when drawing the object on the bottom, we set the last `w.InvalidateRect` parameter (`bErase`) to true to redraw its background but when drawing all the objects that are not on the bottom (`ellipse2` in this case) we set this argument to false.

Here's what this program looks like on the screen:



Here's the complete source file for the *016_graphics.hla2* program:

```
// graphics2-
//
//  This program demonstrates the use of ellipses on a form.

program graphics2;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

?@NoDisplay     := true;
?@NoStackAlign  := true;

#includeOnce( "stdlib.hhf" )
#includeOnce( "howl.hhf" )
```

```
const
    applicationName := "Graphics #2";
    formX           := w.CW_USEDEFAULT; // Let Windows position this guy
    formY           := w.CW_USEDEFAULT;
    formW           := 600;
    formH           := 600;



// Forward declarations for the onClick widgetProcs that we're going to
// call when an event occurs.

proc hideShowEllipse        :widgetProc; @forward;
proc moveEllipse            :widgetProc; @forward;
proc resizeEllipse          :widgetProc; @forward;
proc colorEllipse           :widgetProc; @forward;
proc onClick                :widgetProc; @forward;
proc onQuit                 :widgetProc; @forward;



// Here's the main form definition for the app:

wForm( mainAppWindow );

    var
        showState   :boolean;
        align(4);


    wEllipse
    (
        ellipse1,
        10,
        10,
        200,
        200,
        RGB( 0, 0, 0 ),
        RGB( 255, 255, 255 ),
        howl.transparent_g
    )

    wEllipse
    (
        ellipse2,
        35,
        100,
        200,
        200,
        RGB( 0, 0, 0 ),
        RGB( 255, 255, 255 ),
        howl.transparent_g
    )

    wPushButton
    (
        button2,                     // Field name in mainWindow object
        "Hide ellipse",              // Caption for push button
```

```
    250,                        // x position
    10,                         // y position
    175,                        // width
    25,                         // height
    hideShowEllipse             // initial "on click" event handler
)


wPushButton
(
    button3,                    // Field name in mainWindow object
    "Move Ellipse",             // Caption for push button
    250,                        // x position
    40,                         // y position
    175,                        // width
    25,                         // height
    moveEllipse                 // initial "on click" event handler
)


wPushButton
(
    button4,                    // Field name in mainWindow object
    "Resize Ellipse",               // Caption for push button
    250,                        // x position
    70,                         // y position
    175,                        // width
    25,                         // height
    resizeEllipse                   // initial "on click" event handler
)


wPushButton
(
    button5,                    // Field name in mainWindow object
    "Change Color",             // Caption for push button
    250,                        // x position
    100,                        // y position
    175,                        // width
    25,                         // height
    colorEllipse                    // initial "on click" event handler
)



// Place a quit button in the lower-right-hand corner of the form:

wPushButton
(
    quitButton,             // Field name in mainWindow object
    "Quit",                 // Caption for push button
    450,                    // x position
    525,                    // y position
    125,                    // width
    25,                     // height
    onQuit                  // "on click" event handler
)
```

```
endwForm


// Must invoke the following macro to emit the code generated by
// the wForm macro:

mainAppWindow_implementation();



// The colorEllipse widget proc will change the foreground and background color.

proc colorEllipse:widgetProc;
begin colorEllipse;

    mov( mainAppWindow.ellipse1, esi );
    (type wEllipse_t [esi]).get_lineColor();
    if( eax = RGB( 0, 0, 0 )) then

        (type wEllipse_t [esi]).set_lineColor( RGB( 0, 255, 0 ));
        (type wEllipse_t [esi]).set_fillColor( RGB( 255, 0, 0 ));

    else

        (type wEllipse_t [esi]).set_lineColor( RGB( 0, 0, 0 ));
        (type wEllipse_t [esi]).set_fillColor( RGB( 255, 255, 255 ));

    endif;


    // If we change ellipse1, we've got to force a redraw
    // of ellipse2:

    mov( mainAppWindow.ellipse2, esi );
    w.InvalidateRect( (type wEllipse_t [esi]).handle, NULL, false);

end colorEllipse;



// The resizeEllipse widget proc will resize label1 between widths 150 and 200.

proc resizeEllipse:widgetProc;
begin resizeEllipse;

    mov( mainAppWindow.ellipse1, esi );
    (type wLabel_t [esi]).get_width();
    if( eax = 200 ) then

        stdout.put( "Resizing ellipse to width/height 150" nl );
        (type wEllipse_t [esi]).resize( 150, 150 );

    else

        stdout.put( "Resizing label to width/height 200" nl );
        (type wEllipse_t [esi]).resize( 200, 200 );

    endif;
```

```
    // If we resize ellipse1, we've got to force a redraw
    // of ellipse2:

    mov( mainAppWindow.ellipse2, esi );
    w.InvalidateRect( (type wEllipse_t [esi]).handle, NULL, false);


end resizeEllipse;



// The moveEllipse widget proc will move label
// between y positions 10 and 40.

proc moveEllipse:widgetProc;
begin moveEllipse;

    mov( mainAppWindow.ellipse1, esi );
    (type wEllipse_t [esi]).get_y();
    if( eax = 10 ) then

        stdout.put( "Moving ellipse to y-position 40" nl );
        (type wEllipse_t [esi]).set_y( 40 );

    else

        stdout.put( "Moving ellipse to y-position 10" nl );
        (type wEllipse_t [esi]).set_y( 10 );

    endif;


    // If we move ellipse1, we've got to force a redraw
    // of ellispe1 and ellipse2 in the proper order:

    w.InvalidateRect( (type wEllipse_t [esi]).handle, NULL, true);
    mov( mainAppWindow.ellipse2, esi );
    w.InvalidateRect( (type wEllipse_t [esi]).handle, NULL, false);


end moveEllipse;



// The hideShowEllipse widget proc will hide and show ellipse1.

proc hideShowEllipse:widgetProc;
begin hideShowEllipse;

    mov( thisPtr, esi );
    if( mainAppWindow.showState ) then

        (type wPushButton_t [esi]).set_text( "Hide Ellipse" );
        mov( false, mainAppWindow.showState );
        stdout.put( "Showing ellipse 1" nl );

        mov( mainAppWindow.ellipse1, esi );
        (type wEllipse_t [esi]).show();
```

```
    else

        (type wPushButton_t [esi]).set_text( "Show Ellipse" );
        mov( true, mainAppWindow.showState );
        stdout.put( "Hiding ellipse 1" nl );

        mov( mainAppWindow.ellipse1, esi );
        (type wEllipse_t [esi]).hide();

    endif;

    // If we hide or show ellipse1, we've got to force a redraw
    // of ellipse2:

    mov( mainAppWindow.ellipse2, esi );
    w.InvalidateRect( (type wEllipse_t [esi]).handle, NULL, false);

end hideShowEllipse;




// Here's the onClick event handler the graphic object:

proc onClick:widgetProc;
begin onClick;

    stdout.put( "Clicked on graphic object" nl );
    mov( thisPtr, esi );
    (type wEllipse_t [esi]).set_lineColor( RGB( 0, 0, 255 ));
    (type wEllipse_t [esi]).set_fillColor( RGB( 255, 0, 255 ));


    // If we change the color of ellipse1, we've got to force a redraw
    // of ellipse2:

    mov( mainAppWindow.ellipse2, esi );
    w.InvalidateRect( (type wEllipse_t [esi]).handle, NULL, false);


end onClick;




// Here's the onClick event handler for our quit button on the form.
// This handler will simply quit the application:

proc onQuit:widgetProc;
begin onQuit;

    // Quit the app:

    w.PostQuitMessage( 0 );

end onQuit;
```

```
// We'll use the main application form's onCreate method to initialize
// the various buttons on the form.
//
// This could be done in appStart, but better to leave appStart mainly
// as boilerplate code. Also, putting this code here allows us to use
// "this" to access the mainAppWindow fields (a minor convenience).

method mainAppWindow_t.onCreate;
begin onCreate;

    // Initialize the showState data field:

    mov( false, this.showState );

    // Install onClick handler for the graphic object:

    push( esi );
    mov( (type mainAppWindow_t [esi]).ellipse1, esi );
    (type wEllipse_t [esi]).set_onClick( &onClick );
    pop( esi );
    mov( (type mainAppWindow_t [esi]).ellipse2, esi );
    (type wEllipse_t [esi]).set_onClick( &onClick );

end onCreate;



///////////////////////////////////////////////////////////////////////////////
//
//
// The following is mostly boilerplate code for all apps (about the only thing
// you would change is the size of the main app's form)
//
//
///////////////////////////////////////////////////////////////////////////////
//
// When the main application window closes, we need to terminate the
// application. This overridden method handles that situation.  Notice the
// override declaration for onClose in the wForm declaration given earlier.
// Without that, mainAppWindow_t would default to using the wVisual_t.onClose
// method (which does nothing).

method mainAppWindow_t.onClose;
begin onClose;

    // Tell the winmain main program that it's time to terminate.
    // Note that this message will (ultimately) cause the appTerminate
    // procedure to be called.

    w.PostQuitMessage( 0 );


end onClose;
```

```
// When the application begins execution, the following procedure
// is called.  This procedure must create the main
// application window in order to kick off the execution of the
// GUI application:

procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

    mainAppWindow.create_mainAppWindow
    (
        applicationName,        // Window title
        w.WS_EX_CONTROLPARENT,  // Need this to support TAB control selection
        w.WS_OVERLAPPEDWINDOW,  // Style
        NULL,                   // No parent window
        formX,                  // x-coordinate for window.
        formY,                  // y-coordinate for window.
        formW,                  // Width
        formH,                  // Height
        howl.bkgColor_g,        // Background color
        true                    // Make visible on creation
    );
    mov( esi, pmainAppWindow ); // Save pointer to main window object.
    pop( esi );

end appStart;



// appTerminate-
//
//  Called when the application is quitting, giving the app a chance
// to clean up after itself.
//
// Note that this is called *after* the mainAppWindow_t.onClose method
// executes (indeed, mainAppWindow_t.onClose, by posting the quit message,
// is what actually causes the program to begin terminating, which leads
// to the execution of this procedure).

procedure appTerminate;
begin appTerminate;

    // Clean up the main application's form.
    // Note that this will recursively clean up all the widgets on the form.

    mainAppWindow.destroy();

end appTerminate;


// appException-
//
// Gives the application the opportunity to clean up before
```

```
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;



// The main program for a HOWL application must simply
// call the HowlMainApp procedure.

begin graphics2;

    // Set up the background and transparent colors that the
    // form will use when registering the window_t class:

    w.GetSysColor( w.COLOR_MENU );
    mov( eax, howl.bkgColor_g );
    or( $FF00_0000, eax );
    mov( eax, howl.transparent_g );
    w.CreateSolidBrush( howl.bkgColor_g );
    mov( eax, howl.bkgBrush_g );

    HowlMainApp();

    // Delete the brush we created earlier:

    w.DeleteObject( howl.bkgBrush_g );

end graphics2;
```

Pie Wedges and Pie Graphs

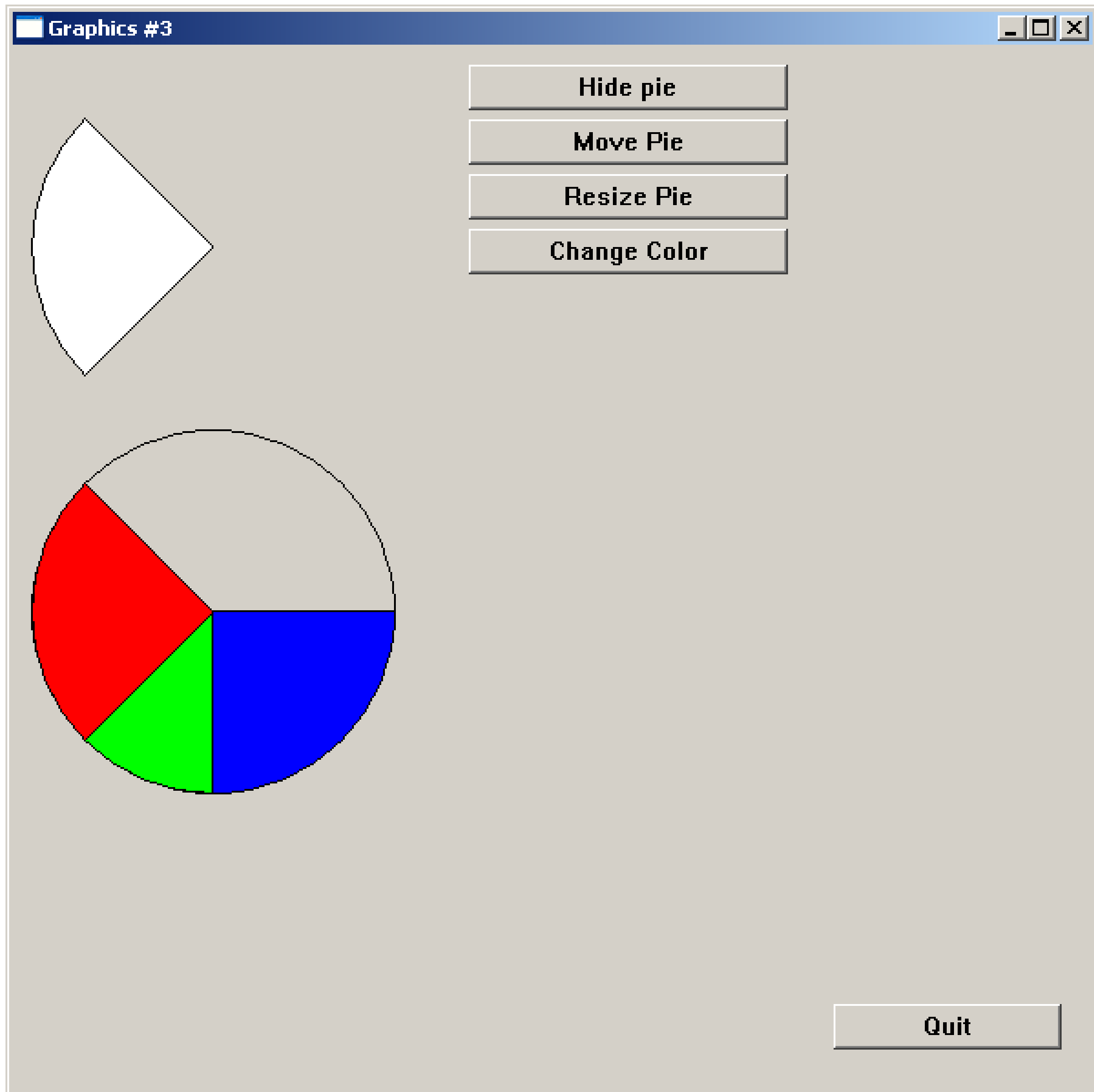
The next sample program we're going to look at, *017_graphics3.hla*, demonstrates drawing pie graph sections on a form. This program draws four pie wedges and a circle (ellipse). The pie1 object is the usual graphics object that the program manipulates to demonstrate various object activities and the remaining three pie wedges and circle demonstrate how to draw a pie graph using the HOWL background transparency feature. This example is structurally similar to the previous two in this tutorial, so let's start by looking at the new code in the HDL section of the program:

```
wPie
(
    pie1,
    10,
    10,
    200,
    200,
    45.0,
    135.0,
    RGB( 0, 0, 0 ),
    RGB( 255, 255, 255 ),
```

```
    howl.bkgColor_g
)


wEllipse
(
    circle1,
    10,
    210,
    200,
    200,
    RGB( 0, 0, 0 ),
    howl.bkgColor_g,
    howl.bkgColor_g
)


wPie
(
    pie2,
    10,
    210,
    200,
    200,
    45.0,
    135.0,
    RGB( 0, 0, 0 ),
    RGB( 255, 0, 0 ),
    howl.transparent_g
)


wPie
(
    pie3,
    10,
    210,
    200,
    200,
    135.0,
    180.0,
    RGB( 0, 0, 0 ),
    RGB( 0, 255, 0 ),
    howl.transparent_g
)

wPie
(
    pie4,
    10,
    210,
    200,
    200,
    180.0,
    270.0,
    RGB( 0, 0, 0 ),
    RGB( 0, 0, 255 ),
    howl.transparent_g
)
```

Notice how the last three `wPie` objects and the `wEllipse` (circle) objects completely overlap. Because of their transparent backgrounds (except for the circle which is drawn at the bottom of the pile), these wedges fill up separate sections of a pie graph chart.

Here's the program's output on the screen:



Because `pie1` doesn't overlap any of the other objects, we don't have to redraw the various pie wedges and circles whenever we manipulate the `pie1` object. Therefore, we avoid some of the software gymnastics present in the last example. However, because the pie wedges are initially draw with an erased background, drawing them over the top of one another wipes out the images just drawn. Therefore, we have to modify the onCreate method so that it redraws the circle and wedges in the proper order:

```
method mainAppWindow_t.onCreate;
begin onCreate;

    // Initialize the showState data field:

    mov( false, this.showState );

    // Install onClick handler for the graphic object:

    mov( (type mainAppWindow_t [esi]).pie1, esi );
    (type wPie_t [esi]).set_onClick( &onClick );

    // Redraw the last three pie wedges without redrawing
    // their backgrounds so that they overlap:

    mov( mainAppWindow.pie2, edi );
    (type wPie_t [esi]).show();

    mov( mainAppWindow.pie3, esi );
    (type wPie_t [esi]).show();

    mov( mainAppWindow.pie4, edi );
    (type wPie_t [esi]).show();


end onCreate;
```

Note that this example calls the show method rather than the `w.InvalidateRect` procedure. The `show` method does the same task as `w.InvalidateRect` with false as the last argument.

Here's the complete code for the *017_graphics3.hla* source file:

```
// graphics3-
//
//  This program demonstrates the use of pie wedges on a form.

program graphics3;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

?@NoDisplay     := true;
?@NoStackAlign  := true;

#includeOnce( "stdlib.hhf" )
#includeOnce( "howl.hhf" )

const
    applicationName := "Graphics #3";
    formX           := w.CW_USEDEFAULT; // Let Windows position this guy
    formY           := w.CW_USEDEFAULT;
    formW           := 600;
    formH           := 600;



// Forward declarations for the onClick widgetProcs that we're going to
// call when an event occurs.
```

```
proc hideShowPie            :widgetProc; @forward;
proc movePie                :widgetProc; @forward;
proc resizePie              :widgetProc; @forward;
proc colorPie               :widgetProc; @forward;
proc onClick                :widgetProc; @forward;
proc onQuit                 :widgetProc; @forward;



// Here's the main form definition for the app:

wForm( mainAppWindow );

    var
        showState   :boolean;
        align(4);


    wPie
    (
        pie1,
        10,
        10,
        200,
        200,
        45.0,
        135.0,
        RGB( 0, 0, 0 ),
        RGB( 255, 255, 255 ),
        howl.bkgColor_g
    )


    wEllipse
    (
        circle1,
        10,
        210,
        200,
        200,
        RGB( 0, 0, 0 ),
        howl.bkgColor_g,
        howl.bkgColor_g
    )


    wPie
    (
        pie2,
        10,
        210,
        200,
        200,
        45.0,
        135.0,
        RGB( 0, 0, 0 ),
        RGB( 255, 0, 0 ),
        howl.transparent_g
```

```
)


wPie
(
    pie3,
    10,
    210,
    200,
    200,
    135.0,
    180.0,
    RGB( 0, 0, 0 ),
    RGB( 0, 255, 0 ),
    howl.transparent_g
)

wPie
(
    pie4,
    10,
    210,
    200,
    200,
    180.0,
    270.0,
    RGB( 0, 0, 0 ),
    RGB( 0, 0, 255 ),
    howl.transparent_g
)



wPushButton
(
    button2,                    // Field name in mainWindow object
    "Hide pie",             // Caption for push button
    250,                        // x position
    10,                         // y position
    175,                        // width
    25,                         // height
    hideShowPie             // initial "on click" event handler
)


wPushButton
(
    button3,                    // Field name in mainWindow object
    "Move Pie",             // Caption for push button
    250,                        // x position
    40,                         // y position
    175,                        // width
    25,                         // height
    movePie                 // initial "on click" event handler
)


wPushButton
(
```

```
        button4,                    // Field name in mainWindow object
        "Resize Pie",               // Caption for push button
        250,                        // x position
        70,                         // y position
        175,                        // width
        25,                         // height
        resizePie                   // initial "on click" event handler
    )


    wPushButton
    (
        button5,                    // Field name in mainWindow object
        "Change Color",             // Caption for push button
        250,                        // x position
        100,                        // y position
        175,                        // width
        25,                         // height
        colorPie                    // initial "on click" event handler
    )




    // Place a quit button in the lower-right-hand corner of the form:

    wPushButton
    (
        quitButton,             // Field name in mainWindow object
        "Quit",                 // Caption for push button
        450,                    // x position
        525,                    // y position
        125,                    // width
        25,                     // height
        onQuit                  // "on click" event handler
    )

endwForm


// Must invoke the following macro to emit the code generated by
// the wForm macro:

mainAppWindow_implementation();







// The colorPie widget proc will change the foreground and background color.

proc colorPie:widgetProc;
begin colorPie;

    mov( mainAppWindow.pie1, esi );
    (type wPie_t [esi]).get_lineColor();
```

```
    if( eax = RGB( 0, 0, 0 )) then

        (type wPie_t [esi]).set_lineColor( RGB( 0, 255, 0 ));
        (type wPie_t [esi]).set_fillColor( RGB( 255, 0, 0 ));

    else

        (type wPie_t [esi]).set_lineColor( RGB( 0, 0, 0 ));
        (type wPie_t [esi]).set_fillColor( RGB( 255, 255, 255 ));

    endif;


end colorPie;



// The resizePie widget proc will resize label1 between widths 150 and 200.

proc resizePie:widgetProc;
begin resizePie;

    mov( mainAppWindow.pie1, esi );
    (type wLabel_t [esi]).get_width();
    if( eax = 200 ) then

        stdout.put( "Resizing pie to width/height 150" nl );
        (type wPie_t [esi]).resize( 150, 150 );

    else

        stdout.put( "Resizing label to width/height 200" nl );
        (type wPie_t [esi]).resize( 200, 200 );

    endif;

end resizePie;



// The movePie widget proc will move label
// between y positions 10 and 40.

proc movePie:widgetProc;
begin movePie;

    mov( mainAppWindow.pie1, esi );
    (type wPie_t [esi]).get_y();
    if( eax = 10 ) then

        stdout.put( "Moving pie to y-position 40" nl );
        (type wPie_t [esi]).set_y( 40 );

    else

        stdout.put( "Moving pie to y-position 10" nl );
        (type wPie_t [esi]).set_y( 10 );

    endif;
```

```
end movePie;



// The hideShowPie widget proc will hide and show pie1.

proc hideShowPie:widgetProc;
begin hideShowPie;

    mov( thisPtr, esi );
    if( mainAppWindow.showState ) then

        (type wPushButton_t [esi]).set_text( "Hide Pie" );
        mov( false, mainAppWindow.showState );
        stdout.put( "Showing pie 1" nl );

        mov( mainAppWindow.pie1, esi );
        (type wPie_t [esi]).show();

    else

        (type wPushButton_t [esi]).set_text( "Show Pie" );
        mov( true, mainAppWindow.showState );
        stdout.put( "Hiding pie 1" nl );

        mov( mainAppWindow.pie1, esi );
        (type wPie_t [esi]).hide();

    endif;

end hideShowPie;





// Here's the onClick event handler the graphic object:

proc onClick:widgetProc;
begin onClick;

    stdout.put( "Clicked on graphic object" nl );
    mov( thisPtr, esi );
    (type wPie_t [esi]).set_lineColor( RGB( 0, 0, 255 ));
    (type wPie_t [esi]).set_fillColor( RGB( 255, 0, 255 ));

end onClick;





// Here's the onClick event handler for our quit button on the form.
// This handler will simply quit the application:

proc onQuit:widgetProc;
begin onQuit;

    // Quit the app:
```

```
    w.PostQuitMessage( 0 );

end onQuit;





// We'll use the main application form's onCreate method to initialize
// the various buttons on the form.
//
// This could be done in appStart, but better to leave appStart mainly
// as boilerplate code. Also, putting this code here allows us to use
// "this" to access the mainAppWindow fields (a minor convenience).

method mainAppWindow_t.onCreate;
begin onCreate;

    // Initialize the showState data field:

    mov( false, this.showState );

    // Install onClick handler for the graphic object:

    mov( (type mainAppWindow_t [esi]).pie1, esi );
    (type wPie_t [esi]).set_onClick( &onClick );

    // Redraw the last three pie wedges without redrawing
    // their backgrounds so that they overlap:

    mov( mainAppWindow.pie2, edi );
    (type wPie_t [esi]).show();

    mov( mainAppWindow.pie3, esi );
    (type wPie_t [esi]).show();

    mov( mainAppWindow.pie4, edi );
    (type wPie_t [esi]).show();


end onCreate;



///////////////////////////////////////////////////////////////////////////////
//
//
// The following is mostly boilerplate code for all apps (about the only thing
// you would change is the size of the main app's form)
//
//
///////////////////////////////////////////////////////////////////////////////
//
// When the main application window closes, we need to terminate the
// application. This overridden method handles that situation.  Notice the
// override declaration for onClose in the wForm declaration given earlier.
// Without that, mainAppWindow_t would default to using the wVisual_t.onClose
// method (which does nothing).
```

```
method mainAppWindow_t.onClose;
begin onClose;

    // Tell the winmain main program that it's time to terminate.
    // Note that this message will (ultimately) cause the appTerminate
    // procedure to be called.

    w.PostQuitMessage( 0 );


end onClose;






// When the application begins execution, the following procedure
// is called.  This procedure must create the main
// application window in order to kick off the execution of the
// GUI application:

procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

    mainAppWindow.create_mainAppWindow
    (
        applicationName,        // Window title
        w.WS_EX_CONTROLPARENT,  // Need this to support TAB control selection
        w.WS_OVERLAPPEDWINDOW,  // Style
        NULL,                   // No parent window
        formX,                  // x-coordinate for window.
        formY,                  // y-coordinate for window.
        formW,                  // Width
        formH,                  // Height
        howl.bkgColor_g,        // Background color
        true                    // Make visible on creation
    );
    mov( esi, pmainAppWindow ); // Save pointer to main window object.
    pop( esi );

end appStart;



// appTerminate-
//
//  Called when the application is quitting, giving the app a chance
// to clean up after itself.
//
// Note that this is called *after* the mainAppWindow_t.onClose method
// executes (indeed, mainAppWindow_t.onClose, by posting the quit message,
// is what actually causes the program to begin terminating, which leads
// to the execution of this procedure).
```

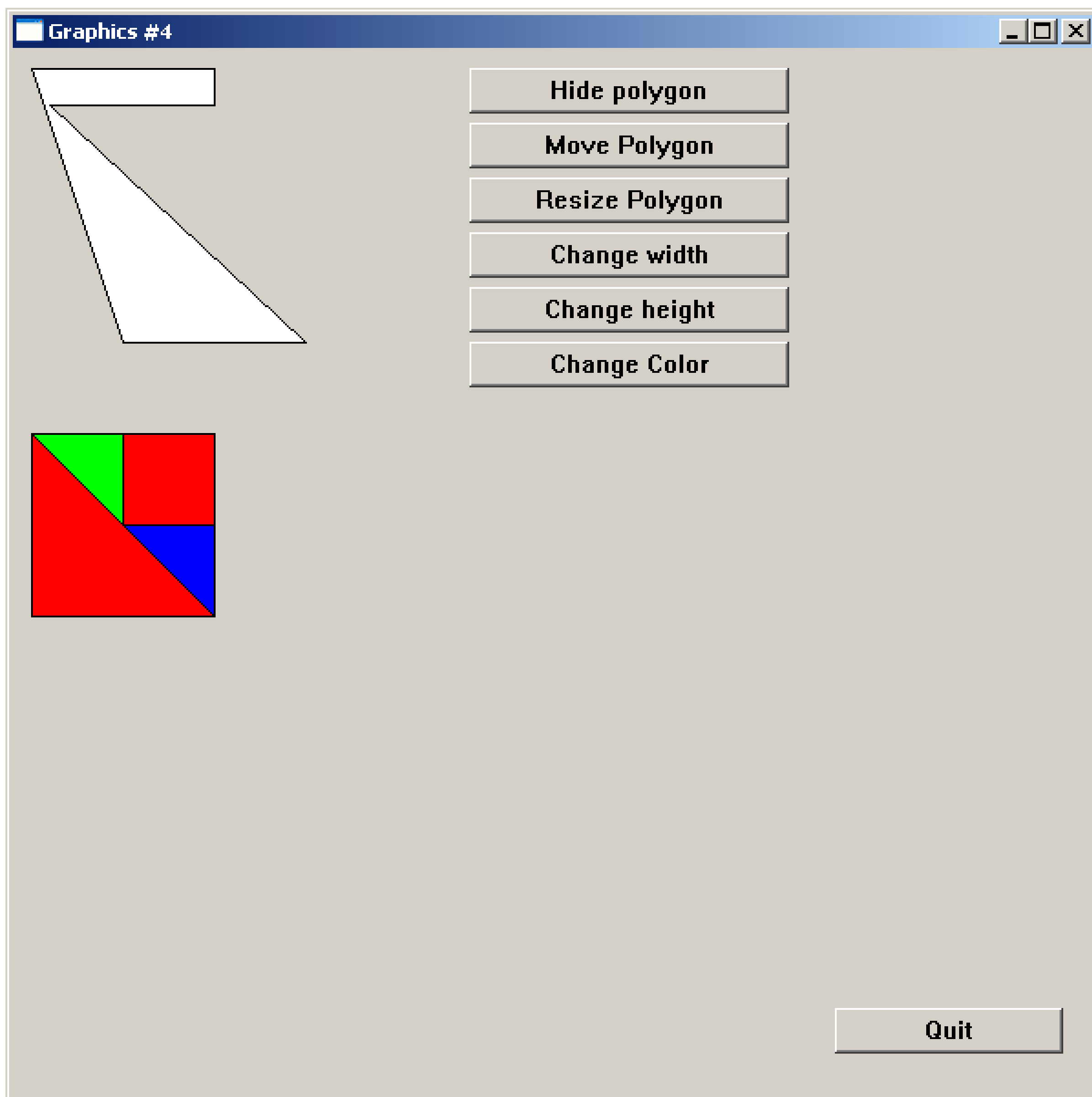
```
procedure appTerminate;
begin appTerminate;

    // Clean up the main application's form.
    // Note that this will recursively clean up all the widgets on the form.

    mainAppWindow.destroy();

end appTerminate;


// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;



// The main program for a HOWL application must simply
// call the HowlMainApp procedure.

begin graphics3;

    // Set up the background and transparent colors that the
    // form will use when registering the window_t class:

    w.GetSysColor( w.COLOR_MENU );
    mov( eax, howl.bkgColor_g );
    or( $FF00_0000, eax );
    mov( eax, howl.transparent_g );
    w.CreateSolidBrush( howl.bkgColor_g );
    mov( eax, howl.bkgBrush_g );

    HowlMainApp();

    // Delete the brush we created earlier:

    w.DeleteObject( howl.bkgBrush_g );

end graphics3;
```

Polygon Example

The next sample program we're going to look at, *018_graphics4.hla*, demonstrates drawing polygons on a form. This program draws four polygons. The polygon1 object is the usual graphics object that the program manipulates to demonstrate various object activities and the remaining three polygons demonstrate how to overlap polygons using the HOWL background transparency feature. This example is structurally similar to the previous three in this tutorial, so let's start by looking at the new code in the HDL section of the program:

```
wPolygon
(
    polygon1,
    10,
    10,
    200,
    200,
    RGB( 0, 0, 0 ),
    RGB( 255, 255, 255 ),
    howl.bkgColor_g
)

wPolygon
(
    polygon2,
    10,
    210,
    200,
    200,
    RGB( 0, 0, 0 ),
    RGB( 255, 0, 0 ),
    howl.transparent_g,
    0, 0,
    100,0,
    100, 100,
    0,100
)

wPolygon
(
    polygon3,
    10,
    210,
    200,
    200,
    RGB( 0, 0, 0 ),
    RGB( 0, 255, 0 ),
    howl.transparent_g,
    0,0,
    50,0,
    50, 50
)

wPolygon
(
    polygon4,
    10,
    210,
    200,
    200,
    RGB( 0, 0, 0 ),
    RGB( 0, 0, 255 ),
    howl.transparent_g,
    50,50,
    100,50,
    100, 100
)
```

Note that the last two polygons overlap the second polygon in this declaration. Here's what the form looks like when *018_graphics4.exe* executes:



The `polygon1` declaration demonstrates the creation of a polygon object without specifying any points for the polygon; the program will have to call `set_points` in order to fully initialize this polygon (this happens in the `onCreate` method). The remaining three polygon definitions (`polygon2..polygon4`) demonstrate polygon declarations that include the points that make up these polygons (a square and two triangles).

About the only other routine that is significantly different from the previous examples is the `onCreate` method. This method contains the code that initializes the points array for `polygon1`. Here is the code for that method:

```
method mainAppWindow_t.onCreate;
readonly
    polyList :w.POINT[] :=
        [
            w.POINT:[0,0],
            w.POINT:[100,0],
            w.POINT:[100, 20],
            w.POINT:[10,20],
            w.POINT:[150,150],
            w.POINT:[50,150]
        ];

begin onCreate;

    // Initialize the showState data field:

    mov( false, this.showState );

    // Set up the polgon points:

    mov( mainAppWindow.polygon1, esi );
    (type wPolygon_t [esi]).set_points( @elements( polyList ), &polyList );


    // Install onClick handler for the graphic object:

    (type wPolygon_t [esi]).set_onClick( &onClick );

    // We have to redraw the last three polygons because they overlap.

    mov( mainAppWindow.polygon2, esi );
    w.InvalidateRect( (type wBase_t [esi]).handle, NULL, false );;

    mov( mainAppWindow.polygon3, esi );
    w.InvalidateRect( (type wBase_t [esi]).handle, NULL, false );;

    mov( mainAppWindow.polygon4, esi );
    w.InvalidateRect( (type wBase_t [esi]).handle, NULL, false );;



end onCreate;
```

Notice how this method redraws the last three polygons to handle the overlap issue (discussed in the previous example).

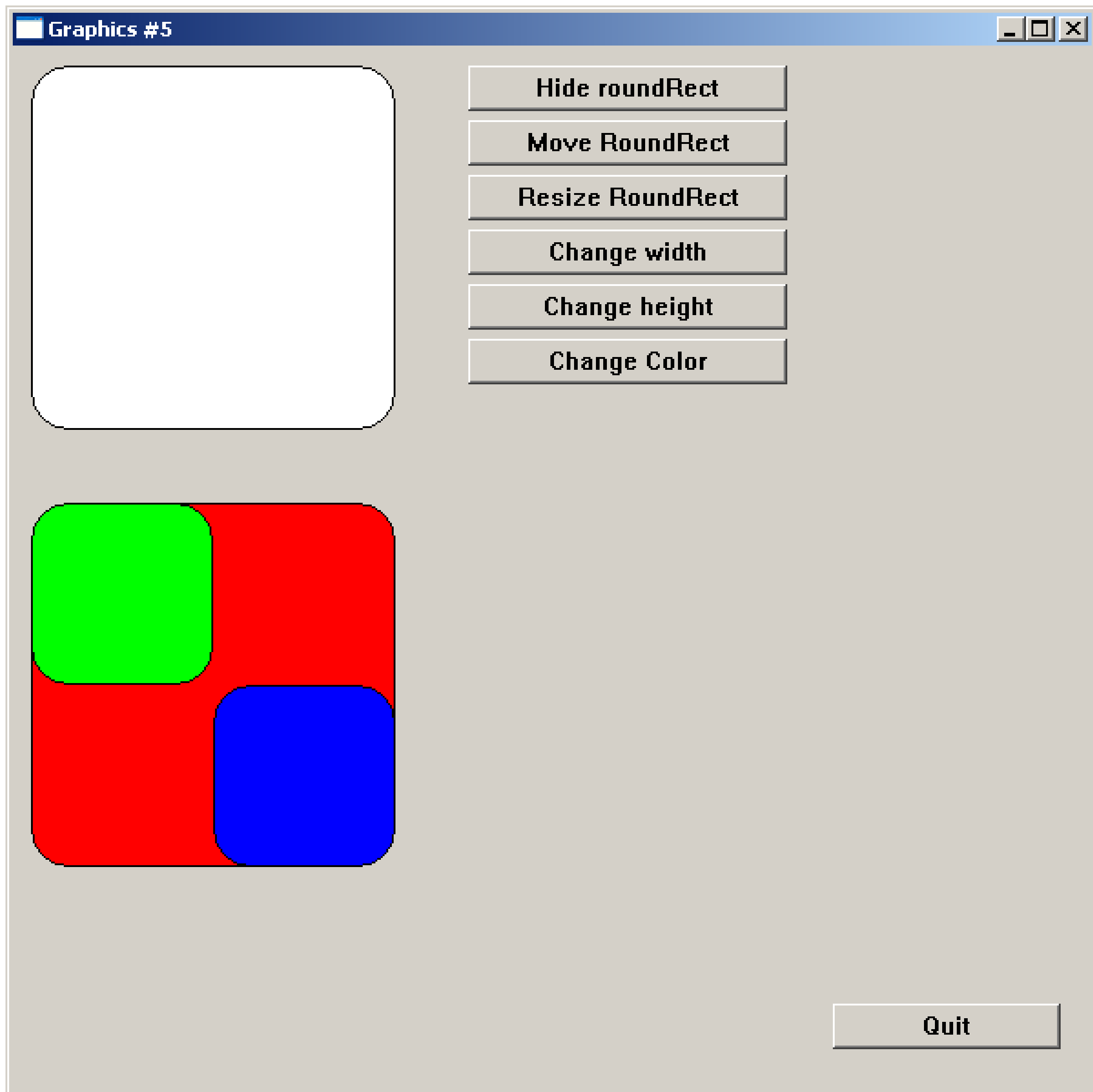
Round Rectangle Example

The last sample program we're going to look at in this tutorial, *019_graphics5.hla*, demonstrates drawing round rectangles on a form. This program draws four round rectangles. The roundRect1 object is the usual graphics object that the program manipulates to demonstrate various object activities and the remaining three round rectangles demonstrate how to overlap round rectangles using the HOWL background transparency feature. This example is structurally similar to the previous three in this tutorial, so let's start by looking at the new code in the HDL section of the program:

```
    wRoundRect
```

```
(
    roundRect1,             // HLA identifier
    10,                     // x
    10,                     // y
    200,                    // width
    200,                    // height
    40,                     // corner width
    40,                     // corner height
    RGB( 0, 0, 0 ),         // outline color (black)
    RGB( 255, 255, 255 ),   // fill color (white)
    howl.bkgColor_g         // background color
)

wRoundRect
(
    roundRect2,
    10,
    250,
    200,
    200,
    40,
    40,
    RGB( 0, 0, 0 ),
    RGB( 255, 0, 0 ),
    howl.transparent_g
)

wRoundRect
(
    roundRect3,
    10,
    250,
    100,
    100,
    40,
    40,
    RGB( 0, 0, 0 ),
    RGB( 0, 255, 0 ),
    howl.transparent_g
)

wRoundRect
(
    roundRect4,
    110,
    350,
    100,
    100,
    40,
    40,
    RGB( 0, 0, 0 ),
    RGB( 0, 0, 255 ),
    howl.transparent_g
)
```

Here's what the program's screen looks like:



Here's the complete source file for *019_graphics5.hla*:

```
 // graphics5-
//
//  This program demonstrates the use of round rectangles on a form.

program graphics5;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

?@NoDisplay     := true;
```

```
?@NoStackAlign  := true;

#includeOnce( "stdlib.hhf" )
#includeOnce( "howl.hhf" )

const
    applicationName := "Graphics #5";
    formX           := w.CW_USEDEFAULT; // Let Windows position this guy
    formY           := w.CW_USEDEFAULT;
    formW           := 600;
    formH           := 600;



// Forward declarations for the onClick widgetProcs that we're going to
// call when an event occurs.

proc hideShowRoundRect      :widgetProc; @forward;
proc moveRoundRect          :widgetProc; @forward;
proc resizeRoundRect        :widgetProc; @forward;
proc changeHeight           :widgetProc; @forward;
proc changeWidth            :widgetProc; @forward;
proc colorRoundRect         :widgetProc; @forward;
proc onClick                :widgetProc; @forward;
proc onQuit                 :widgetProc; @forward;



// Here's the main form definition for the app:

wForm( mainAppWindow );

    var
        showState   :boolean;
        align(4);


    wRoundRect
    (
        roundRect1,             // HLA identifier
        10,                     // x
        10,                     // y
        200,                    // width
```

```
    200,                      // height
    40,                       // corner width
    40,                       // corner height
    RGB( 0, 0, 0 ),           // outline color (black)
    RGB( 255, 255, 255 ),     // fill color (white)
    howl.bkgColor_g           // background color
)

wRoundRect
(
    roundRect2,
    10,
    250,
    200,
    200,
    40,
    40,
    RGB( 0, 0, 0 ),
    RGB( 255, 0, 0 ),
    howl.transparent_g
)

wRoundRect
(
    roundRect3,
    10,
    250,
    100,
    100,
    40,
    40,
    RGB( 0, 0, 0 ),
    RGB( 0, 255, 0 ),
    howl.transparent_g
)

wRoundRect
(
    roundRect4,
    110,
    350,
    100,
    100,
```

```
    40,
    40,
    RGB( 0, 0, 0 ),
    RGB( 0, 0, 255 ),
    howl.transparent_g
)

wPushButton
(
    button2,                    // Field name in mainWindow object
    "Hide roundRect",           // Caption for push button
    250,                        // x position
    10,                         // y position
    175,                        // width
    25,                         // height
    hideShowRoundRect           // initial "on click" event handler
)


wPushButton
(
    button3,                    // Field name in mainWindow object
    "Move RoundRect",           // Caption for push button
    250,                        // x position
    40,                         // y position
    175,                        // width
    25,                         // height
    moveRoundRect               // initial "on click" event handler
)


wPushButton
(
    button4,                    // Field name in mainWindow object
    "Resize RoundRect",         // Caption for push button
    250,                        // x position
    70,                         // y position
    175,                        // width
    25,                         // height
    resizeRoundRect             // initial "on click" event handler
)
```

```
wPushButton
(
    button5,                  // Field name in mainWindow object
    "Change width",           // Caption for push button
    250,                      // x position
    100,                      // y position
    175,                      // width
    25,                       // height
    changeWidth               // initial "on click" event handler
)


wPushButton
(
    button6,                  // Field name in mainWindow object
    "Change height",          // Caption for push button
    250,                      // x position
    130,                      // y position
    175,                      // width
    25,                       // height
    changeHeight              // initial "on click" event handler
)


wPushButton
(
    button7,                  // Field name in mainWindow object
    "Change Color",           // Caption for push button
    250,                      // x position
    160,                      // y position
    175,                      // width
    25,                       // height
    colorRoundRect            // initial "on click" event handler
)



// Place a quit button in the lower-right-hand corner of the form:

wPushButton
(
    quitButton,           // Field name in mainWindow object
```

```
            "Quit",                 // Caption for push button
            450,                    // x position
            525,                    // y position
            125,                    // width
            25,                     // height
            onQuit                  // "on click" event handler
        )

    endwForm


    // Must invoke the following macro to emit the code generated by
    // the wForm macro:

    mainAppWindow_implementation();
```

```
    // The colorRoundRect widget proc will change the foreground and
background color.

    proc colorRoundRect:widgetProc;
    begin colorRoundRect;

        mov( mainAppWindow.roundRect1, esi );
        (type wRoundRect_t [esi]).get_lineColor();
        if( eax = RGB( 0, 0, 0 )) then

            (type wRoundRect_t [esi]).set_lineColor( RGB( 0, 255, 0 ));
            (type wRoundRect_t [esi]).set_fillColor( RGB( 255, 0, 0 ));

        else

            (type wRoundRect_t [esi]).set_lineColor( RGB( 0, 0, 0 ));
            (type wRoundRect_t [esi]).set_fillColor( RGB( 255, 255, 255 ));

        endif;
```

```
   end colorRoundRect;



   // The resizeRoundRect widget proc will resize label1 between widths 150
and 200.

   proc resizeRoundRect:widgetProc;
   begin resizeRoundRect;

       mov( mainAppWindow.roundRect1, esi );
       (type wRoundRect_t [esi]).get_width();
       if( eax = 200 ) then

           stdout.put( "Resizing roundRect to width/height 150" nl );
           (type wRoundRect_t [esi]).resize( 150, 150 );

       else

           stdout.put( "Resizing label to width/height 200" nl );
           (type wRoundRect_t [esi]).resize( 200, 200 );

       endif;

   end resizeRoundRect;



   // The changeWidth widget proc will set the roundRect between widths 150
and 200.

   proc changeWidth:widgetProc;
   begin changeWidth;

       mov( mainAppWindow.roundRect1, esi );
       (type wRoundRect_t [esi]).get_width();
       if( eax = 200 ) then

           stdout.put( "Resizing roundRect to width 150" nl );
           (type wRoundRect_t [esi]).set_width( 150 );

       else
```

```
            stdout.put( "Resizing label to width 200" nl );
            (type wRoundRect_t [esi]).set_width( 200 );

        endif;

    end changeWidth;



    // The changeHeight widget proc will set the roundRect between Heights
150 and 200.

    proc changeHeight:widgetProc;
    begin changeHeight;

        mov( mainAppWindow.roundRect1, esi );
        (type wRoundRect_t [esi]).get_height();
        if( eax = 200 ) then

            stdout.put( "Resizing roundRect to height 150" nl );
            (type wRoundRect_t [esi]).set_height( 150 );

        else

            stdout.put( "Resizing label to height 200" nl );
            (type wRoundRect_t [esi]).set_height( 200 );

        endif;

    end changeHeight;



    // The moveRoundRect widget proc will move label
    // between y positions 10 and 40.

    proc moveRoundRect:widgetProc;
    begin moveRoundRect;

        mov( mainAppWindow.roundRect1, esi );
        (type wRoundRect_t [esi]).get_y();
        if( eax = 10 ) then
```

```
        stdout.put( "Moving roundRect to y-position 40" nl );
        (type wRoundRect_t [esi]).set_y( 40 );

    else

        stdout.put( "Moving roundRect to y-position 10" nl );
        (type wRoundRect_t [esi]).set_y( 10 );

    endif;

end moveRoundRect;



// The hideShowRoundRect widget proc will hide and show roundRect1.

proc hideShowRoundRect:widgetProc;
begin hideShowRoundRect;

    mov( thisPtr, esi );
    if( mainAppWindow.showState ) then

        (type wPushButton_t [esi]).set_text( "Hide RoundRect" );
        mov( false, mainAppWindow.showState );
        stdout.put( "Showing roundRect 1" nl );

        mov( mainAppWindow.roundRect1, esi );
        (type wRoundRect_t [esi]).show();

    else

        (type wPushButton_t [esi]).set_text( "Show RoundRect" );
        mov( true, mainAppWindow.showState );
        stdout.put( "Hiding roundRect 1" nl );

        mov( mainAppWindow.roundRect1, esi );
        (type wRoundRect_t [esi]).hide();

    endif;

end hideShowRoundRect;
```

```
// Here's the onClick event handler the graphic object:

proc onClick:widgetProc;
begin onClick;

    stdout.put( "Clicked on graphic object" nl );
    mov( thisPtr, esi );
    (type wRoundRect_t [esi]).set_lineColor( RGB( 0, 0, 255 ));
    (type wRoundRect_t [esi]).set_fillColor( RGB( 255, 0, 255 ));

end onClick;




// Here's the onClick event handler for our quit button on the form.
// This handler will simply quit the application:

proc onQuit:widgetProc;
begin onQuit;

    // Quit the app:

    w.PostQuitMessage( 0 );

end onQuit;





// We'll use the main application form's onCreate method to initialize
// the various buttons on the form.
//
// This could be done in appStart, but better to leave appStart mainly
// as boilerplate code. Also, putting this code here allows us to use
// "this" to access the mainAppWindow fields (a minor convenience).

method mainAppWindow_t.onCreate;
begin onCreate;
```

```
        // Initialize the showState data field:

        mov( false, this.showState );

        // Install onClick handler for the graphic object:

        mov( (type mainAppWindow_t [esi]).roundRect1, esi );
        (type wRoundRect_t [esi]).set_onClick( &onClick );


    end onCreate;




    ///////////////////////////////////////////////////////////////////////////
///////
    //
    //
    // The following is mostly boilerplate code for all apps (about the only
thing
    // you would change is the size of the main app's form)
    //
    //
    ///////////////////////////////////////////////////////////////////////////
///////
    //
    // When the main application window closes, we need to terminate the
    // application. This overridden method handles that situation.  Notice
the
    // override declaration for onClose in the wForm declaration given
earlier.
    // Without that, mainAppWindow_t would default to using the
wVisual_t.onClose
    // method (which does nothing).

    method mainAppWindow_t.onClose;
    begin onClose;

        // Tell the winmain main program that it's time to terminate.
        // Note that this message will (ultimately) cause the appTerminate
        // procedure to be called.

        w.PostQuitMessage( 0 );
```

```
    end onClose;




    // When the application begins execution, the following procedure
    // is called.  This procedure must create the main
    // application window in order to kick off the execution of the
    // GUI application:

    procedure appStart;
    begin appStart;

        push( esi );

        // Create the main application window:

        mainAppWindow.create_mainAppWindow
        (
            applicationName,        // Window title
            w.WS_EX_CONTROLPARENT,  // Need this to support TAB control
selection
            w.WS_OVERLAPPEDWINDOW,  // Style
          NULL,               // No parent window
            formX,                  // x-coordinate for window.
            formY,                  // y-coordinate for window.
            formW,                  // Width
            formH,                  // Height
            howl.bkgColor_g,        // Background color
            true                    // Make visible on creation
        );
        mov( esi, pmainAppWindow ); // Save pointer to main window object.
        pop( esi );

    end appStart;



    // appTerminate-
    //
    //  Called when the application is quitting, giving the app a chance
```

```
   // to clean up after itself.
   //
   // Note that this is called *after* the mainAppWindow_t.onClose method
   // executes (indeed, mainAppWindow_t.onClose, by posting the quit
message,
   // is what actually causes the program to begin terminating, which leads
   // to the execution of this procedure).

   procedure appTerminate;
   begin appTerminate;

       // Clean up the main application's form.
       // Note that this will recursively clean up all the widgets on the
form.

       mainAppWindow.destroy();

   end appTerminate;


   // appException-
   //
   // Gives the application the opportunity to clean up before
   // aborting when an unhandled exception comes along:

   procedure appException( theException:dword in eax );
   begin appException;

       raise( eax );

   end appException;



   // The main program for a HOWL application must simply
   // call the HowlMainApp procedure.

   begin graphics5;

       // Set up the background and transparent colors that the
       // form will use when registering the window_t class:

       w.GetSysColor( w.COLOR_MENU );
       mov( eax, howl.bkgColor_g );
```

```
    or( $FF00_0000, eax );
    mov( eax, howl.transparent_g );
    w.CreateSolidBrush( howl.bkgColor_g );
    mov( eax, howl.bkgBrush_g );

    HowlMainApp();

    // Delete the brush we created earlier:

    w.DeleteObject( howl.bkgBrush_g );

end graphics5;
```