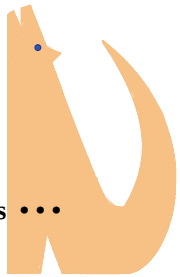
Chapter 5

• *Discovering the Possibilities* • • •



Graphical Display Tricks

Chapter Overview

In the last chapter you learned a number of graphical display techniques. In this chapter, you learn several graphical display tricks that can be used with graphical displays to give your presentations a professional look and feel.

Specifically, you will learn:

- How to provide cursor interaction with your graphical display
- How to erase annotation from your graphical display
- How to draw “rubberband” symbols on your graphical display
- How to use the Z graphics buffer for graphical display tricks

Using the Cursor with Graphical Displays

One of the reasons data is displayed visually is so the user can interact with it in one way or another. One way users like to interact with data is to use the mouse cursor to select or annotate portions of their data. This kind of interaction is easily accomplished in IDL using the *Cursor* command.

To see how the *Cursor* command works, load the *Time Serie*s data set with the *LoadData* command like this:

```
IDL> curve = LoadData(1)
```

Display the curve by typing these commands:

```
IDL> Window, XSize=400, YSize=400
IDL> LoadCT, 0
IDL> TvLCT, 255, 255, 0, 1
IDL> Plot, curve
```

The *Cursor* command accepts two arguments. These must be variables in which the position of the cursor when a mouse button is pushed is recorded. The *Cursor* command requires that the cursor be located in the current graphics window. (This is the window pointed to by the system variable *!D.Window*.) For example, if you type this

command, IDL will be waiting for you to move your cursor into the current graphics window (window index 0 if you typed the commands above) and click the mouse button down. When you do, IDL will put the position of the cursor into the variables *xLocation* and *yLocation*. Type:

```
IDL> Cursor, xLocation, yLocation
```

If you print these values out, you will see that the values are given in data coordinate space. That is the *xLocation* values will be between 0 and 100 and the *yLocation* values will be between 0 and 30. (At least they will be if you clicked the mouse inside the plot boundaries. What happens if you do not?) The *Cursor* command returns data coordinate positions by default.

```
IDL> Print, xLocation, yLocation
```

When Is the Cursor Position Returned?

It would appear from the commands above that the cursor position is returned when the mouse button is pushed down, but this is not always the case. In fact, when the *Cursor* command reports its position is determined by keywords to the *Cursor* command. These keywords are:

Change — The position is reported when there is a *change* in the cursor's position, or when the user *moves* the cursor.

Down — The position is returned when the mouse button is pushed *down*.

NoWait — The position is returned immediately when the *Cursor* command is executed. There is no delay or wait for mouse buttons. This keyword is sometimes used in loops when objects are being moved on the display.

Up — The position is returned not when the mouse button is clicked down, but when it comes *up* or is released.

Wait — The *Cursor* command *waits* for the button to be pressed to report its position. As long as the button is pressed down, this keyword causes the *Cursor* command to act as though it had been invoked with the *NoWait* keyword. This is the default behavior for the *Cursor* command.

Be careful to use the proper keyword with the *Cursor* command, especially if you are using the *Cursor* command in a loop. Users sometimes get into the habit of thinking that the default behavior for the *Cursor* command is to only report back when the cursor is clicked *down*. It is not. The default behavior is to *wait* for a click then act as if a *no wait* were in effect. In a loop this difference can be critical.

Which Mouse Button Was Used with the Cursor?

In addition to setting the behavior of the cursor, you also sometimes want to know which mouse button was used to respond to the *Cursor* command. For example, you may want to do one thing if the right mouse button was used and something different if the left mouse button was used in response to the *Cursor* command.

You can determine which button was used with the *Cursor* command by examining the *Button* field of the *!Mouse* system variable. (Older versions of IDL used the value of the *!Err* system variable for this same purpose.) This field is an integer bit map. Valid values for the *Button* field and their meanings are as follows:

!Mouse.Button = 0 No button has currently been used.

!Mouse.Button = 1 The left mouse button was used with the *Cursor* command.

!Mouse.Button = 2 The middle mouse button was used.

!Mouse.Button = 4 The right mouse button was used.

Annotating Graphics Output with the Cursor

One way the *Cursor* command might be used is to allow the user to interactively place symbols on a line plot. For example, type the commands below exactly as they appear here. When you hit the final carriage return, click your mouse five times in the current graphics window. Five symbols will be placed in the window. (If you make a typing mistake in the code below, start again with the first line when you correct it.) Type:

```
IDL> FOR j=0, 4 DO BEGIN $
IDL> Cursor, xloc, yloc, /Down & $
IDL> PlotS, xloc, yloc, PSym=4, SymSize=2, Color=1 & ENDFOR
```

Drawing a Box

You might want to select a portion of the display and draw a box around it. Here are some commands to select the diagonal corners of a box with the *Cursor* command, draw the box (be sure to draw the box so that it includes a portion of the actual data), and zoom the plot into the box coordinates. First, draw the plot:

```
IDL> Plot, curve
```

Next, use the cursor to select one corner of the box you want to draw. You will want to be sure to click the cursor in the current graphics window. To make sure you know which one that is and that it is not hidden, type:

```
IDL> WShow
```

Now type the first *Cursor* command. Click somewhere inside the plot axes:

```
IDL> Cursor, x1, y1, /Down  ; Select one corner of box.
```

Now type the second *Cursor* command. Click somewhere inside the plot axes:

```
IDL> Cursor, x2, y2, /Down  ; Select diagonal corner of box.
```

The coordinates returned from the Cursor commands above are in *data* coordinate space. Draw the box like this:

```
IDL> PlotS, [x1,x1, x2 x2,x1], [y1,y2,y2,y1,y1], Color=1
```

Your output will look something like the illustration in Figure 58, although the actual box on your plot will depend on where you clicked in the window.

To zoom into this portion of the plot, you will have to be sure the box coordinates are ordered properly. This is necessary because you may have first selected the lower-right corner of the box and then the upper-left, in which case *x1* will be greater than *x2*. You can imagine other scenarios as well. To account for all of them, type:

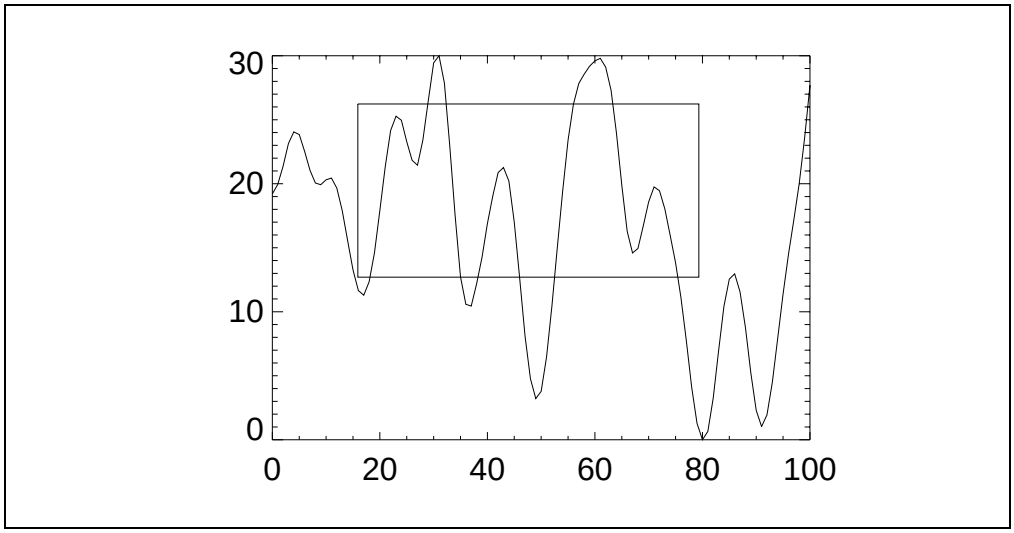


Figure 58: A line plot with a box drawn around the portion of the data. The box coordinates were selected with the Cursor command and the box drawn with the PlotS command.

```
IDL> xmin = Min([x1,x2], Max=xmax)
IDL> ymin = Min([y1,y2], Max=ymax)
```

Finally, you are ready to zoom into the portion of the data enclosed by the box. In addition to setting the data ranges properly, you also have to set the *[XY]Style* keywords. Do you know why? If you don't, try the command below without these two keywords. What happens?

```
IDL> Plot, curve, XRange=[xmin,xmax], YRange=[ymin,ymax], $
       XStyle=1, YStyle=1
```

Using the Cursor with Images

Normally when you are working with image data and using the *Cursor* command, you want the cursor locations in device coordinates rather than data coordinates. This is because there is usually a simple relationship (most of the time one-to-one) between the device coordinate and the equivalent location in the image. To see how this works, open the 360 by 360 *World Elevation* data set with the *LoadData* command, like this:

```
IDL> image = LoadData(7)
```

Display the image and load some colors like this:

```
IDL> topColor = !D.N_Colors-1
IDL> LoadCT, 3, NColors=!D.N_Colors-1
IDL> TvLCT, 255, 255, 0, topColor
IDL> Window, XSize=360, YSize=360
IDL> TV, BytScl(image, Top=!D.N_Colors-2)
```

Now use the cursor to select a particular column and row in the image. Notice the *Device* keyword in the *Cursor* and *PlotS* commands. This is to be sure the coordinates are in *device* coordinates and not *data* coordinates. Draw a cross-hair at that location. (Be sure to click in the image window after you type the *Cursor* command.) Type:

```
IDL> s = Size(image)
IDL> Cursor, col, row, /Device ; Click in the window!
IDL> PlotS, [col, col], [0, s(2)], /Device, Color=topColor
IDL> PlotS, [0, s(1)], [row, row], /Device, Color=topColor
```

Notice how easy it is to access the data in the image in that particular column and row. For example, you can easily plot the column and row data profiles for the image, like this:

```
IDL> Window, 1, XSize=500, YSize=300
IDL> !P.Multi = [0, 2, 1]
IDL> Plot, image(col, *), Title='Row Profile'
IDL> Plot, image(*, row), Title='Column Profile'
IDL> !P.Multi = 0
IDL> WSet, 0
```

Your output should look similar to the illustration in Figure 59.

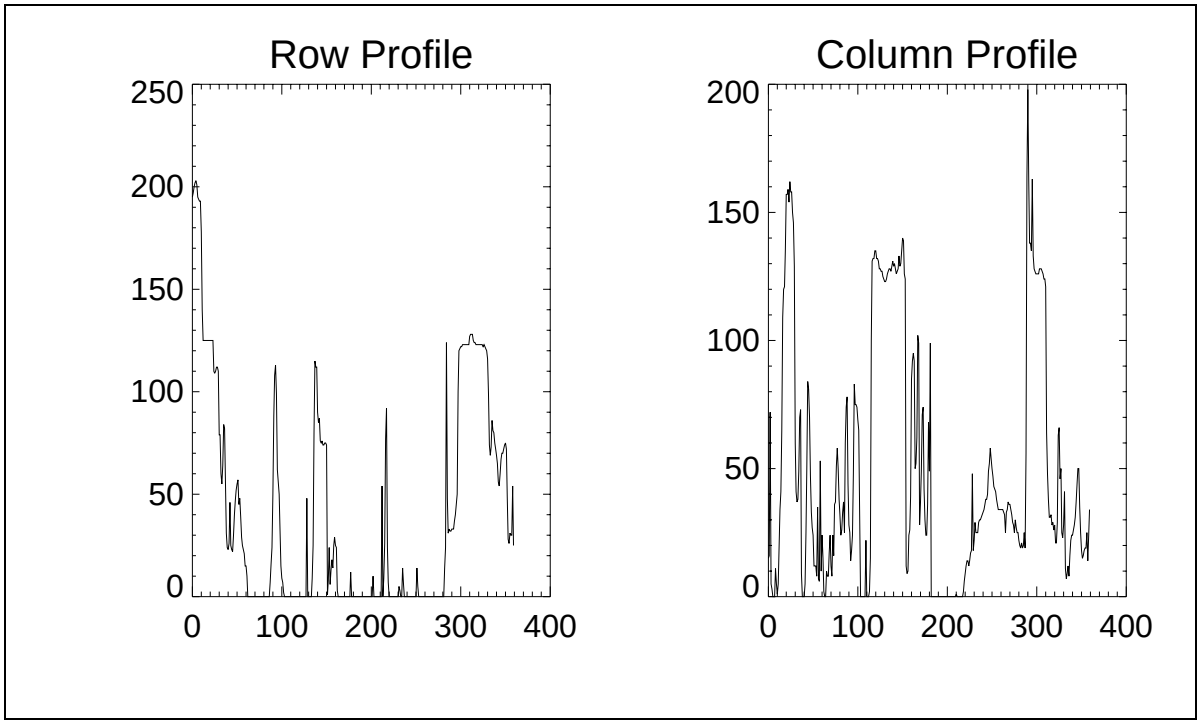


Figure 59: The column and row profiles of the image in the column and row selected with the Cursor command.

Using the Cursor in Loops

Sometimes you want to use the *Cursor* command in a loop. For example, you might want to know the value of each individual image pixel as you select it with the cursor. Here is a simple loop that continues until you click the *right* or *middle* mouse button to get out of it. Open a text editor and type these commands *exactly* as you see here.

```
topColor = !D.N_Colors-1
LoadCT, 3, NColors=!D.N_Colors-1
TvLCT, 255, 255, 0, topColor
TV, BytScl(image, Top=!D.N_Colors-2)
```

```
!Mouse.Button = 1
REPEAT BEGIN
   Cursor, col, row, /Down, /Device
   Print, 'Pixel Value: ', image(col, row)
ENDREP UNTIL !Mouse.Button NE 1
END
```

Save the file as *loop1.pro*. (This file is among the files you downloaded to use with this book.) To compile and run this little main-level program, type:

```
IDL> .RUN loop1
```

Move your cursor into the image window and start clicking with the left mouse button. You will see the image pixel values printed out in your log window until you use some button other than the left in the image window.

What happens if you use other keywords besides *Down* with the *Cursor* command? Experiment a little and find out.

Erasing Annotation From the Display

Using the cursor to place annotations on the graphical display the way you have been doing tends to beg the question: "But, how do I *erase* what I just put there!" There are two preferred ways to erase annotations. I call these the *exclusive OR* method and the *device copy* method. Of the two, the device copy method gives more professional looking results, in my opinion. Both are presented here, but the focus will be on the device copy technique.

The "Exclusive OR" Method of Erasing Annotation

The exclusive OR method of erasing annotation works on the basis of what are called *graphics functions*. A graphics function is a bit-level operation on two numbers. These numbers are associated with the pixel that is already on the display (this is the so-called *destination* pixel) and the pixel you wish to put in that same location (this is the so-called *source* pixel).

Normally, the graphics function IDL uses is called *SOURCE*. In this graphics function, IDL ignores the value of the destination pixel and just puts the value of the source pixel at the pixel location. But if the graphics function is changed to *XOR* (exclusive OR) IDL does a bit-wise comparison of the bits of the destination pixel with the source pixel. This has the effect of "flipping" the bit values of the destination pixel. In other words, if the binary representation of the pixel value is 01100101. Then after the XOR operation, the binary representation of the pixel value is 10011010.

(The true XOR story is more complicated than this, because it only really works this way if IDL has 256 colors in contiguous locations in the color lookup table, and this is seldom the case. Most people just think of XOR mode as drawing in the "opposite" color and leave it at that. In true *XOR* mode you could predict what color you would be drawing with, but this is not true with this mode under most circumstances. This is the reason most professional IDL programmers prefer the device copy technique.)

The graphics function in effect at any particular time is set with the *Device* command and the *Set_Graphics_Function* keyword. *SOURCE* mode is graphics function 3; *XOR*

mode is graphics function 6. At the moment IDL is in its default *SOURCE* mode. While you are in this mode, redisplay the image in the image window. Type:

```
IDL> TV, BytScl(image, Top=!D.N_Colors-2)
```

Now, select *XOR* mode, like this:

```
IDL> Device, Set_Graphics_Function=6
```

You will draw a box on the image like this:

```
IDL> PlotS, [0.2, 0.2, 0.8, 0.8, 0.2], Color=topColor, $
        [0.2, 0.8, 0.8, 0.2, 0.2], /Normal
```

You will notice that the line of the box is not yellow, as you might have expected, but is instead a sort of multicolored hue, although it shows up reasonably well. The underlying pixels have been “flipped” in this graphics function.

To erase the box, all you have to do is flip the underlying pixel values back to their original values. This is easily done by issuing the *PlotS* command again, like this:

```
IDL> PlotS, [0.2, 0.2, 0.8, 0.8, 0.2], Color=topColor, $
        [0.2, 0.8, 0.8, 0.2, 0.2], /Normal
```

You can issue the above command over and over, making the box appear and disappear at will. Before you go on, be sure you set your graphics function back to *SOURCE* mode, like this:

```
IDL> Device, Set_Graphics_Function=3
```

You can easily take advantage of graphics functions in your IDL programs. For example, open the *loop1.pro* main-level program you wrote earlier and modify it to look like this. Here you are going to draw a large cross-hair at each image location as you click in the image window. Save this program as *loop2.pro*. Type:

```
topColor = !D.N_Colors-1
LoadCT, 3, NColors=!D.N_Colors-1
TvLCT, 255, 255, 0, topColor
TV, BytScl(image, Top=!D.N_Colors-2)
!Mouse.Button = 1

   ; Go into XOR mode.

Device, Set_Graphics_Function=6

   ; Get initial cursor location. Draw cross-hair.

Cursor, col, row, /Device, /Down
PlotS, [col,col], [0,360], /Device, Color=topColor
PlotS, [0,360], [row,row], /Device, Color=topColor
Print, 'Pixel Value: ', image(col, row)

   ; Loop.

REPEAT BEGIN

      ; Get new cursor location.

   Cursor, colnew, rownew, /Down, /Device

      ; Erase old cross-hair.

   PlotS, [col,col], [0,360], /Device, Color=topColor
   PlotS, [0,360], [row,row], /Device, Color=topColor
```

```
      Print, 'Pixel Value: ', image(colnew, rownew)

         ; Draw new cross-hair.

      PlotS, [colnew,colnew], [0,360], /Device, Color=topColor
      PlotS, [0,360], [rownew,rownew], /Device, Color=topColor

         ; Update coordinates.

      col = colnew
      row = rownew
   ENDREP UNTIL !Mouse.Button NE 1

      ;Erase the final cross-hair.

   PlotS, [col,col], [0,360], /Device, Color=topColor
   PlotS, [0,360], [row,row], /Device, Color=topColor

      ; Restore normal graphics function.

   Device, Set_Graphics_Function=3
   END
```

Save the file as *loop2.pro*. (You can find *loop2.pro* among the files you downloaded to use with this book.) To compile and run this main-level program, type:

```
IDL> .RUN loop2
```

Place your cursor in the image window and click several times with your left mouse button. You should see a cross-hair at each cursor location. To exit the program, click the right or middle mouse button.

The "Device Copy" Method of Erasing Annotation

The device copy technique uses *pixmap windows* to erase annotation that you put on the display. A pixmap window is identical to any other IDL graphics window, except that it doesn't exist on your display. In fact, it exists in the video RAM of your display device. In other words, it exists in memory. But in every other respect, it is like a normal IDL graphics window: it is created with the *Window* command, it is made active with the *WSet* command, it is deleted with the *WDelete* command, etc. You draw graphics in a pixmap window in exactly the same way you draw graphics in a normal IDL graphics windows (e.g., with *Plot*, *Surface*, *TV*, and other graphics output commands).

The device copy technique involves copying a rectangular area from one window (called the *source* window) and pasting the rectangle into another window (called the *destination* window). The source and destination window can sometimes be the same window, as you will see in a moment. You see an illustration of the device copy technique in Figure 60.

The actual copying is done with the *Device* command and the *Copy* keyword (hence, the name of the technique). The general form of the command is this:

```
Device, Copy=[sx, sy, col, row, dx, dy, sourceWindowID]
```

In this command, the elements of the *Copy* keyword are:

sx, sy — The device coordinates of the lower-left corner of the rectangle in the *source* window. (The source window is the window the rectangle is being copied from.)

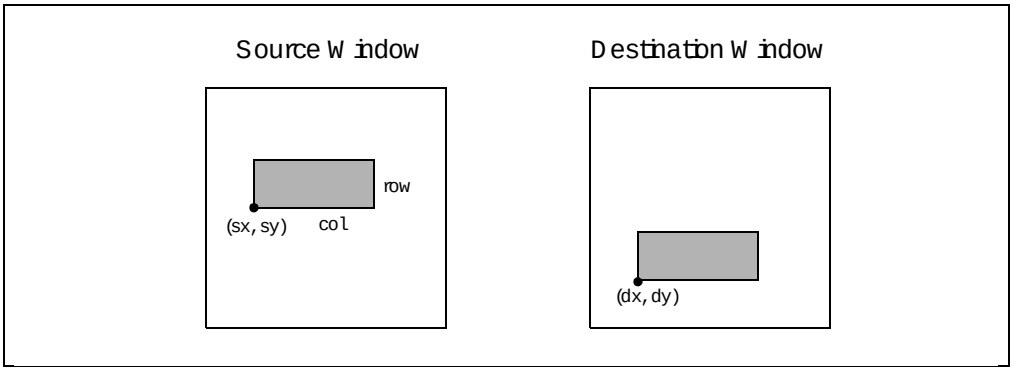


Figure 60: The device copy technique involves copying a rectangular portion of the source window into a location in the destination window. In practice entire windows may be copied or the source and destination window can be the same window.

col — The number of columns to copy in the source window. This is the width of the rectangle.

row — The number of rows to copy in the source window. This is the height of the rectangle.

dx, dy — The device coordinates of the lower-left corner of the rectangle in the *destination* window. (The *destination* window is the window the rectangle is being copied to. The *destination* window is *always* the current graphics window.)

sourceWindowID — This is the window index number of the source window. The rectangle is copied from this window into the current graphics window (which is identified by the *!D.Window* system variable). The source window can be the current graphics window, but it is more often a window other than the current graphics window. It is often a pixmap window.

To see how this works, create a pixmap window and display the image in it. Pixmap windows are created with the *Window* command and the *Pixmap* keyword, like this:

```
IDL> Window, 1, /Pixmap, XSize=360, YSize=360
IDL> TV, BytScl(image, Top=!D.N_Colors-2)
```

Notice that you had no visual clue that anything happened when you typed these commands. This is because the pixmap window exists only in RAM, not on the display. To be sure there is something in this window, open a third, regular window and try to copy the contents of the pixmap window into it. If your third window looks like your image window, you have typed the commands correctly. Type:

```
IDL> Window, 2, XSize=360, YSize=360
IDL> Device, Copy=[0, 0, 360, 360, 0, 0, 1]
```

Notice that you copied the entire contents of the pixmap window into this new window. This is similar to just re-displaying the image in the new window, except that it is several orders of magnitude faster. It is not unusual to copy the entire contents of a

pixmap window into a display window, even if you just have to “repair” a portion of the display window.

Delete the last two windows you created (including the pixmap window), like this:

```
IDL> WDelete, 1, 2
```

It is important to remember to delete pixmap windows when you are finished with them. They do take up memory that you may want to use for something else. Some window managers allocate a fixed amount of memory for pixmap windows. Others use virtual memory if your pixmap window exceeds the capacity of the video RAM. X-terminals have notoriously little memory for pixmap windows.

To see how the device copy technique works in practice, modify the main-level program you wrote earlier and named *loop2.pro*. You may want to copy that program to another file and name it *loop3.pro*. Make the modifications shown below.

```
topColor = !D.N_Colors-1
LoadCT, 3, NColors=!D.N_Colors-1
TvLCT, 255, 255, 0, topColor
TV, BytScl(image, Top=!D.N_Colors-2)
!Mouse.Button = 1

   ; Create a pixmap window and display image in it.

Window, 1, /Pixmap, XSize=360, YSize=360
TV, BytScl(image, Top=!D.N_Colors-2)

   ;Make the display window the current graphics window.

WSet, 0

   ; Get initial cursor location. Draw cross-hair.

Cursor, col, row, /Device, /Down
PlotS, [col,col], [0,360], /Device, Color=topColor
PlotS, [0,360], [row,row], /Device, Color=topColor
Print, 'Pixel Value: ', image(col, row)

   ; Loop.

REPEAT BEGIN

      ; Get new cursor location.

   Cursor, colnew, rownew, /Down, /Device

      ; Erase old cross-hair.

   Device, Copy=[0, 0, 360, 360, 0, 0, 1]
   Print, 'Pixel Value: ', image(colnew, rownew)

      ; Draw new cross-hair.

   PlotS, [colnew,colnew], [0,360], /Device, Color=topColor
   PlotS, [0,360], [rownew,rownew], /Device, Color=topColor
ENDREP UNTIL !Mouse.Button NE 1

   ;Erase the final cross-hair.

Device, Copy=[0, 0, 360, 360, 0, 0, 1]
END
```

Save the file as *loop3.pro*. (This file is among those you downloaded to use with this book.) To compile and run this main-level program, type:

```
IDL> .RUN loop3
```

Place your cursor in the image window and click several times with your left mouse button. To exit the program, click the right or middle mouse button. Notice that the cross-hairs are drawn in a yellow color.

Delete the pixmap window before you move on to the next exercise. Type:

```
IDL> WDelete, 1
```

Drawing a Rubberband Box

The device copy technique is an excellent one for drawing rubberband selection boxes and other shapes on the display. (A rubber band box is a box that has one fixed corner and one dynamic corner that follows the cursor around.) In fact, your *Loop3* program can be easily modified. Copy the *loop3.pro* program into a file named *rubberbox.pro*. (The *loop3.pro* file is among those you downloaded to use with this book.) Make the modifications below to see how easy it is to create a rubberband box.

```
topColor = !D.N_Colors-1
LoadCT, 3, NColors=!D.N_Colors-1
TvLCT, 255, 255, 0, topColor
TV, BytScl(image, Top=!D.N_Colors-2)
!Mouse.Button = 1

   ; Create a pixmap window and display image in it.

Window, 1, /Pixmap, XSize=360, YSize=360
TV, BytScl(image, Top=!D.N_Colors-2)

   ;Make the display window the current graphics window.

WSet, 0

   ; Get initial cursor location (static corner of box).

Cursor, sx, sy, /Device, /Down

   ; Loop.

REPEAT BEGIN

      ; Get new cursor location (dynamic corner of box).

   Cursor, dx, dy, /Wait, /Device

      ; Erase the old box.

   Device, Copy=[0, 0, 360, 360, 0, 0, 1]

      ; Draw the new box.

   PlotS, [sx,sx,dx,dx,sx], [sy,dy,dy,sy,sy], /Device, $
      Color=topColor
ENDREP UNTIL !Mouse.Button NE 1

   ;Erase the final box.

Device, Copy=[0, 0, 360, 360, 0, 0, 1]
END
```

To run this program, type:

```
IDL> .RUN rubberbox
```

Delete the pixmap window before you move on to the next exercise. Type:

```
IDL> WDelete, 1
```

Graphics Window Scrolling

Another good application of the device copy technique is to implement window scrolling. In this example you will scroll the image in the graphics display window with the device copy technique. The image will scroll four columns at a time from left to right. The algorithm you will use is this: (1) Copy the last four rows on the right of the window into a small pixmap window that is just four columns wide and 360 rows tall, then (2) Move the entire contents of the display window (minus the four rows you just copied) over to the right four rows in the same window (i.e., the source window and the destination window are identical), and finally (3) Copy the contents of the pixmap window into the first four rows on the left of the display window. Open a text editor and type the commands below. Name your program *scroll.pro*. (This program is among those you downloaded to use with this book.) Type:

```
   ; Open a pixmap window 4 columns wide.
Window, 1, /Pixmap, XSize=4, YSize=360
FOR j=0,360/4 DO BEGIN
   ; Copy four columns on right of display into pixmap.
Device, Copy=[356, 0, 4, 360, 0, 0, 0]
   ; Make the display window the active window.
WSet, 0
   ; Move window contents over 4 columns.
Device, Copy=[0, 0, 356, 360, 4, 0, 0]
   ; Copy pixmap contents into display window on left.
Device, Copy=[0, 0, 4, 360, 0, 0, 1]
ENDFOR
END
```

To run this program, type:

```
IDL> .Run scroll
```

The program scrolls once. To run it again, type:

```
IDL> .Go
```

Can you modify the program to make it keep scrolling until you stop it?

Delete the pixmap window before you move on to the next exercise. Type:

```
IDL> WDelete, 1
```

Graphics Display Tricks in the Z-Graphics Buffer

You can think of the Z-graphics buffer in IDL as a three-dimensional box in which 3D objects can be deposited without regard to their "solidity." The box has the ability to keep track of the "depth" of an object in a 16-bit depth buffer. One side of the box is a projection plane. You can think of rays of light going through each pixel in the projection plane and eventually encountering a solid object in the box. The pixel value the light ray encounters is the value that is "projected" onto the projection plane. In this way, the Z-graphics buffer can take care of hidden surface and line removal automatically. You see an illustration of this concept in Figure 61.

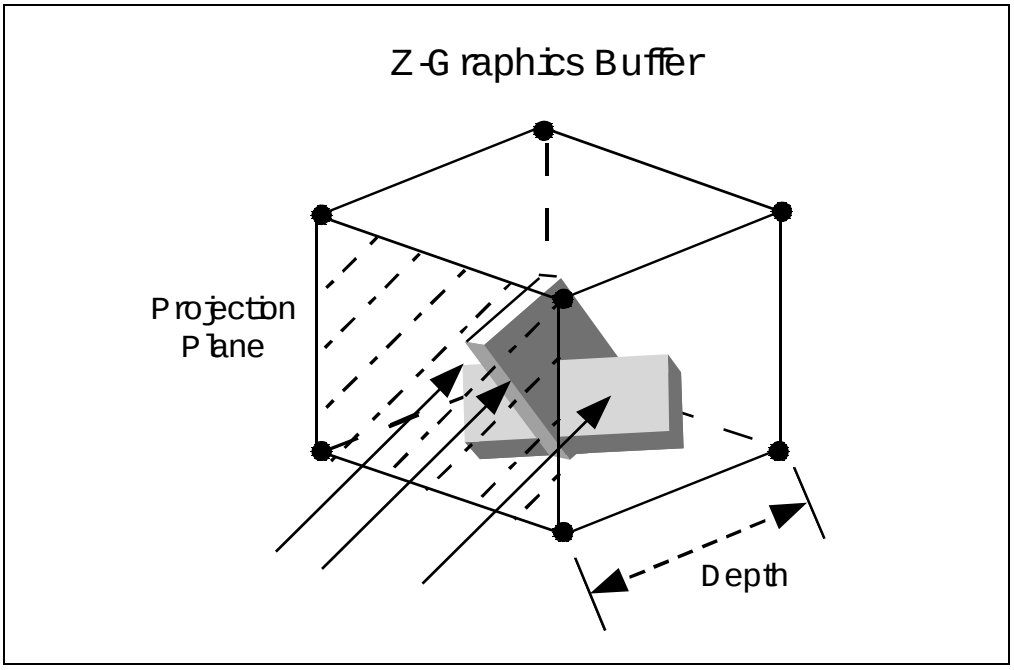


Figure 61: The Z-graphics buffer can be thought of as a 3D box that keeps track of depth information. Rays hit the objects in the Z-graphic buffer and their pixel values are projected back onto the projection plane.

The idea is that once you load your objects into the 3D box, you take a "snap-shot" or picture of the projection plane. This is the 2D projection of the 3D objects in the box. Objects that are behind other objects will not be shown. (This behavior can be modified by the *Transparent* keyword to some IDL graphics commands, as you will see.) The snap-shot is, in effect, a screen dump of the projection plane take with the *TVRD* command.

The Z-Graphics Buffer Implementation

The Z-graphics buffer is implemented in software in IDL as another graphics output device, similar to the PostScript device or your normal X, Win, or Mac device. Thus, to write to the Z-graphics buffer you must make it the current graphics output device with the *Set_Plot* command. As with other graphics output devices, the Z-graphics buffer is configured with the *Device* command and appropriate keywords.

Two keywords that are often used with the Z-graphics buffer are *Set_Colors* and *Set_Resolution*. The keywords are defined like this:

Set_Colors The number of colors in the Z-graphics buffer. By default the Z-graphics buffer uses 256 colors. This is seldom the number of colors in your IDL session. If you want the output from the Z-graphics buffer to have the same number of colors as your display device, you will need to set this keyword.

Set_Resolution The projection plane of the Z-graphics buffer is normally set to 640 pixels wide and 480 pixels high. You should set the resolution of the Z-graphics buffer to the size of the graphics window you want to display the output in.

A Z-Graphics Buffer Example: Two Surfaces

To see how the Z-graphic buffer works, create two objects named *peak* and *saddle*, like this. (The commands to implement this example can be found in the file *two-surf.pro* that you downloaded to use with this book.)

```
IDL> peak = Shift(Dist(20, 16), 10, 8)
IDL> peak = Exp( - (peak / 5) ^ 2)
IDL> saddle = Shift(peak, 6, 0) + Shift(peak, -6, 0) / 2B
```

You are going to combine these two 3D objects in the Z-graphics buffer, but first you might like to see what these objects look like on their own. You will display them with different color tables in two windows. First, load a blue and red color table in different portions of the color lookup table. Type:

```
IDL> colors = !D.N_Colors/2
IDL> LoadCT, 1, NColors=colors
IDL> LoadCT, 3, NColors=colors, Bottom=colors-1
```

Create a window and display the shaded surface plot of the first object. Notice that the *Set_Shading* command is used to restrict the shading values to a particular portion of the color lookup table. Type:

```
IDL> Window, 1, XSize=300, YSize=300
IDL> Set_Shading, Values=[0,colors-1]
IDL> Shade_Surf, peak, ZRange=[0.0, 1.2]
```

Display the second object in its own display window. Use a different portion of the color lookup table for the shading parameters. Type:

```
IDL> Window, 2, XSize=300, YSize=300
IDL> Set_Shading, Values=[colors, 2*colors-1]
IDL> Shade_Surf, saddle, ZRange=[0.0, 1.2]
```

Make the Z-Graphics Buffer the Current Device

To combine these two objects in the Z-graphics buffer, you must make the Z-graphics buffer the current graphics display device. This is done with the *Set_Plot* command. The *Copy* keyword copies the current color table into the Z-buffer. Be sure to save the name of your current graphics display device, so you can get back to it easily. Type:

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'Z', /Copy
```

Configure the Z-Graphics Buffer

Next, you must configure the Z-graphics device to your specifications. In this case, you want to restrict the number of colors and you want to make the buffer resolution equivalent to the size of the current graphics display windows. Type:

```
IDL> Device, Set_Colors=2*colors, Set_Resolution=[300,300]
```

Load the Objects into the Z-Graphics Buffer

Now, put the two objects into the Z-graphics buffer. Notice that you don't see anything happening as you type these commands. The output is going into the Z-graphics buffer in memory, not to the display device. Type:

```
IDL> Set_Shading, Values=[0,colors-1]
IDL> Shade_Surf, peak, ZRange=[0.0, 1.2]
IDL> Set_Shading, Values=[colors, 2*colors-1]
IDL> Shade_Surf, saddle, ZRange=[0.0, 1.2], /NoErase
```

Take a Picture of the Projection Plane

Next, take a "snap-shot" of the projection plane. This is done with the *TVRD* command, like this:

```
IDL> picture = TVRD()
```

Display the Result on the Display Device

Finally, return to your display device, open a new window to display the result, and display the "picture," like this:

```
IDL> Set_Plot, thisDevice
IDL> Window, 3, XSize=300, YSize=300
IDL> TV, picture
```

Your output should look similar to the illustration in Figure 62.

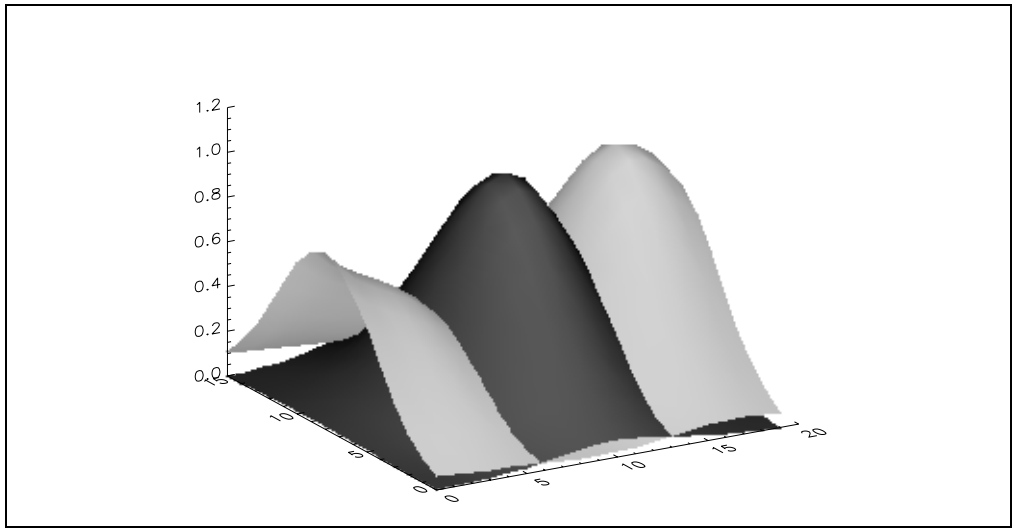


Figure 62: The Z-graphics buffer can be used to combine 3D objects with hidden surface removal performed automatically.

Some Z-Graphics Buffer Oddities

Look carefully at the output in, say, windows 1 and 3. Look particularly at the axes labels. Notice that the axes annotations are written slightly larger in window 3, the output that came from the Z-graphics buffer. The Z-graphics buffer for some reason uses a different default character size than IDL does when it is displaying graphics in a display window.

This simple fact can cause you untold hours of difficulty if you don't realize it when you are setting up a 3D coordinate space in the Z-graphics buffer and combining IDL graphics commands in the Z-graphics buffer. This is primarily because plot margins are based on default character size and plot margins are not the same on the display and in the Z-graphics buffer. They are *almost* the same. But it is the "almost" that will drive you crazy.

A rule of thumb that is *enormously* helpful, is to always set the *!P.Charsize* system variable if you are going to be doing graphics in the Z-graphics buffer. For example, like this:

```
IDL> !P.Charsize = 1.0
```

Just to give you an example, look at the illustration in Figure 62. The axes on this plot were not created in the Z-graphics buffer because then they would have been rendered in screen resolution (i.e., as an image) and I wanted to render them in PostScript resolution. If the *!P.Charsize* keyword had *not* been set prior to rendering the shaded surfaces and adding the axes later, it would have been impossible to get the axes to line up in the correct position in the final output.

Warping Images with the Z-Graphics Buffer

One of the more powerful techniques to use with the Z-graphics buffer is to use it to display slices of a 3D data set. This is possible because of the ability to warp images onto a polygon plane with the Z-graphics buffer. To see how this works, open the 80 by 100 by 57 3D *MRI Head Scan* data set with the *LoadData* command, like this:

```
IDL> head = LoadData(8)
```

You may want to open a journal file to capture these commands as you type them, since there are many of them and you have to get them exactly right. A journal file will enable you to make changes and re-run the commands easily. (This journal file has already been created for you and can be found in the file *warping.pro* that you downloaded to use with this book.)

```
IDL> Journal, 'warping.pro'
```

In general, the dimensions or size of a variable are found with the *Size* command in IDL. You need to know the sizes of the three dimensions in order to define the proper image plane. In this case, the "size" is going to be one less than the true size of the dimension, because you want to use this number as an index into an array, and IDL uses zero-based indexing. Type:

```
IDL> s = Size(head)
IDL> xs = s(1) - 1
IDL> ys = s(2) - 1
IDL> zs = s(3) - 1
```

Suppose you want to display the three orthogonal slices at the center of this data set, say through the 3D point (40, 50, 27). You can define these points, like this:

```
IDL> xpt = 40
IDL> ypt = 50
IDL> zpt = 27
```

Next, you want to construct the individual polygons that describe these three image slices or planes. In this case, each polygon will be a simple rectangle with four points (the corners of the rectangle). Each point in the rectangle will be described by an (x,y,z) triple. Another way to say this is that each plane will be a 3 by 4 polygon. You can type this:

```
IDL> xplane = [ [xpt, 0, 0], [xpt, 0, zs], [xpt, ys, zs], $
       [xpt, ys, 0] ]
IDL> yplane = [ [0, ypt, 0], [0, ypt, zs], [xs, ypt, zs], $
       [xs, ypt, 0] ]
IDL> zplane = [ [ 0, 0, zpt], [xs, 0, zpt], [xs, ys, zpt], $
       [0, ys, zpt] ]
```

The next step is to get the image data that will correspond to each image plane. This is done easily by using array subscripts in IDL. Type:

```
IDL> ximage = head(xpt, *, *)
IDL> yimage = head(*, ypt, *)
IDL> zimage = head(*, *, zpt)
```

Notice that these images are all 3D images (one dimension is a 1). What you want are 2D images associated with each image plane, so you must reformat these 3D images into 2D images with the *Reform* command, like this. In this case, the *Reform* command reformats the image into an 80 by 100 by 1 image. When the final dimension in an array is 1, IDL discards it. The result here is an 80 by 100 image.

```
IDL> ximage = Reform(ximage)
IDL> yimage = Reform(yimage)
IDL> zimage = Reform(zimage)
```

To display these images properly, you want to make sure they are scaled properly into the number of colors on your display. It is important to scale them in relation to the entire data set. Scale the data and load colors like this:

```
IDL> minData = Min(head, Max=maxData)
IDL> topColor = !D.N_Colors-2
IDL> LoadCT, 5, NColors=!D.N_Colors-1
IDL> TvLCT, 255, 255, 255, topColor+1
IDL> ximage = BytScl(ximage, Top=topColor, Max=maxData, $
       Min=minData)
IDL> yimage = BytScl(yimage, Top=topColor, Max=maxData, $
       Min=minData)
IDL> zimage = BytScl(zimage, Top=topColor, Max=maxData, $
       Min=minData)
```

Next, you are ready to set up the Z-graphics buffer. The *Erase* command will erase whatever may have been left in the buffer previously. In this case, you are erasing with the a white color, to give more definition to your display. Type:

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'Z'
```

```
IDL> Device, Set_Colors=topColor, Set_Resolution=[400,400]
IDL> Erase, Color=topColor + 1
```

Set up the 3D coordinate space with the *Scale3* command. Here the axes will be labelled with the size of each dimension. Type:

```
IDL> Scale3, XRange=[0,xs], YRange=[0,ys], ZRange=[0,zs]
```

You are finally ready to render the slices in the Z-graphics buffer. You will use the *Polyfill* command for this purpose. The *Pattern* keyword will be set to the image slice you wish to display. The *Image_Coord* keyword contains a list of the image coordinates associated with each vertex of the polygon. The *Image_Interp* keyword specifies that bilinear interpolation occurs rather than nearest neighbor re-sampling as the image is warped into the polygon. The *T3D* keyword ensures that the polygon is represented in 3D space by applying the 3D transformation matrix to the final output. Type:

```
IDL> Polyfill, xplane, /T3D, Pattern=ximage, /Image_Interp, $
        Image_Coord=[ [0,0], [0, zs], [ys, zs], [ys, 0] ]
IDL> Polyfill, yplane, /T3D, Pattern=yimage, /Image_Interp, $
       Image_Coord=[ [0,0], [0, zs], [xs, zs], [xs, 0] ]
IDL> Polyfill, zplane, /T3D, Pattern=zimage, /Image_Interp, $
       Image_Coord=[ [0,0], [xs, 0], [xs, ys], [0, ys] ]
```

Finally, take a snap-shot of the projection plane, and display the result, like this:

```
IDL> picture = TVRD()
IDL> Set_Plot, thisDevice
IDL> Window, XSize=400, YSize=400
IDL> TV, picture
```

If you opened a journal file, close it now:

```
IDL> Journal
```

Your output should look like the illustration in Figure 63. If it doesn't, modify the code in your journal file with a text editor to fix the problem. To re-run the code, save the file and type this:

```
IDL> @warping
```

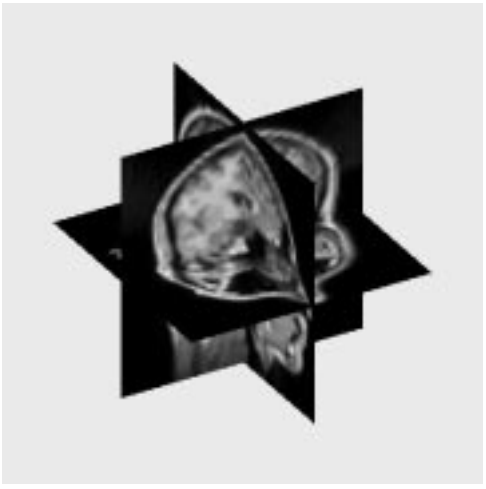


Figure 63: An example of warping image data into planes in the Z-graphics buffer.

Transparency Effects in the Z-Graphics Buffer

Notice that each slice in Figure 63 has quite a lot of black around the outside edges of the slice. This is not part of the image. Rather, it is part of the background noise.

One of the nice features of the Z-graphics buffer is that you can apply transparency effects in it. For example, if you set the *Transparent* keyword on the *Polyfill* command above to roughly 20 or 25, then all those pixels below that value in the image will be transparent. You can see what this looks like by typing this. (You may want to start another journal file. If you closed the last journal file, give this file a new name. Unfortunately, it is not possible to append to a journal file. Call the new journal file *transparent.pro*. You can find a copy of this journal file among the files you downloaded to use with this book.)

```
IDL> Journal, 'transparent'
IDL> Set_Plot, 'Z'
IDL> Erase, Color=topColor + 1
IDL> Polyfill, xplane, /T3D, Pattern=ximage, /Image_Interp, $
       Image_Coord=[ [0,0], [0, zs], [ys, zs], [ys, 0] ], $
       Transparent=25
IDL> Polyfill, yplane, /T3D, Pattern=yimage, /Image_Interp, $
       Image_Coord=[ [0,0], [0, zs], [xs, zs], [xs, 0] ], $
       Transparent=25
IDL> Polyfill, zplane, /T3D, Pattern=zimage, /Image_Interp, $
       Image_Coord=[ [0,0], [xs, 0], [xs, ys], [0, ys] ], $
       Transparent=25
IDL> picture = TVRD()
IDL> Set_Plot, thisDevice
IDL> Window, /Free, XSize=400, YSize=400
IDL> Erase, Color=topColor + 1
IDL> TV, picture
IDL> Journal
```

Combining Z-Graphics Buffer Effects with Volume Rendering

Z-graphics buffer effects are often combined with volume rendering techniques to make vivid visual displays of data. For example, suppose you wanted to create an iso-surface of this data set. (An iso-surface is a surface that has the same value everywhere. It is like a three-dimensional contour plot of the data.) Start a journal file named *isosurface.pro*. (A copy of this journal file is among the files you downloaded to use with this book.)

```
IDL> Journal, 'isosurface.pro'
```

To create an iso-surface, first use the *Shade_Volume* command to create a list of vertices and polygons that describe the surface. In the command below, the *Low* keyword is set so that all the values greater than the iso-surface value will be enclosed by the iso-surface. The variables *vertices* and *polygons* are output variables. They will be used in the subsequent *PolyShade* command to render the surface. A look at the histogram of the head data suggests a value of 50 will be a suitable contouring level. Type:

```
IDL> Plot, Histogram(head), Max_Value=5000
IDL> Shade_Volume, head, 50, vertices, polygons, /Low
```

The iso-surface is rendered with the *PolyShade* command. Be sure to use the 3D transformation that you set up earlier, like this:

```
IDL> Scale3, XRange=[0,xs], YRange=[0,ys], ZRange=[0,zs]
IDL> isosurface = PolyShade(vertices, polygons, /T3D)
IDL> LoadCT, 0, NColors=topColor+1
IDL> TV, isosurface
```

Now, combine this iso-surface with the Z slice through the data set that you took earlier in the Z-graphics buffer. Notice that you are truncating the head data in the Z direction. Type:

```
IDL> Shade_Volume, head(*,*,0:zpt), 50, vertices, $
        polygons, /Low
IDL> isosurface = PolyShade(vertices, polygons, /T3D)
IDL> isosurface(Where(isosurface EQ 0)) = topColor+1
IDL> TvLCT, 70, 70, 70, topColor+1
IDL> Set_Plot, 'Z', /Copy
IDL> TV, isosurface
IDL> Scale3, XRange=[0,xs], YRange=[0,ys], ZRange=[0,zs]
IDL> Polyfill, zplane, /T3D, Pattern=zimage, /Image_Interp, $
       Image_Coord=[ [0,0], [xs, 0], [xs, ys], [0, ys] ], $
       Transparent=25
IDL> picture = TVRD()
IDL> Set_Plot, thisDevice
IDL> TV, picture
IDL> Journal
```

Your output should look like the illustration in Figure 64. If it doesn't, modify your journal file and run it over again by typing this:

```
IDL> @isosurface
```

Figure 64: The isosurface and data slice combined in the Z-graphics buffer.