

Chapter 9. Object graphics

9.1. Introduction

Object graphics is a complete graphics system; it is an alternative to the direct graphics system. It provides an object-oriented interface to OpenGL, a specification for writing cross-platform graphics applications implemented widely amongst modern graphics cards.

The object graphics system is a more modern graphics system than direct graphics. It is a good choice for creating nearly any visualization, but particularly useful for 3-dimensional and interactive visualizations. Its main drawback is that more setup is required to create the hierarchy of objects necessary to display a scene. The iTools provide, amongst other things, a simple procedural interface to object graphics visualizations, making object graphics usable from the command line. A more detailed comparison of the features of object graphics to direct graphics is covered in Section 6.1, “Direct graphics vs. object graphics” [p. 135].

Object graphics can be rendered either using hardware (an OpenGL enabled graphics card) or software (the Mesa OpenGL emulation library). The output when using hardware rendering is highly dependent on how well the graphics card supports OpenGL. Software rendering is more reliable, but is slower. Generally, hardware rendering is tried first and software rendering is used if there are problems.

If the output from the examples in this chapter looks odd, try using software rendering. All the examples in this chapter accept a *RENDERER* keyword that can be set to 1 to force the example to use software rendering. To use software rendering for all object graphics displays by default, use

```
IDL> pref_set, 'IDL_GR_X_RENDERER', 1, /commit
```

on Unix platforms, or

```
IDL> pref_set, 'IDL_GR_WIN_RENDERER', 1, /commit
```

on Windows.

9.2. Displaying an image in object graphics

The first example using object graphics will be displaying an image. To make any object graphics scene, a hierarchy of objects is created; each object in the tree has its own task in the final display. The simplest scene has a view, a model, and a graphics atom connected together in a hierarchy: a view at the top, a model beneath it, and an atom at the bottom.

First, read in an image to display:

```
IDL> peopleFilename = filepath('people.jpg', subdir=['examples', 'data'])
IDL> ali = read_image(peoplesFilename)
IDL> help, ali
ALI          BYTE          = Array[3, 256, 256]
```

Next, let's create the required objects and place them into a hierarchy to display this image data. At the top level, an *IDLgrView* is created to define the coordinate system and other top-level properties of the visualization:

```
IDL> view = obj_new('IDLgrView', viewplane_rect=[0, 0, 256, 256])
```

The *VIEWPLANE_RECT* keyword here specifies that the lower left corner of the view is (0, 0) and that the view is 256 units wide and 256 units tall. The coordinates of the data objects, here in pixel coordinates, must fall in this range to be

visible. There are several techniques to scale the data into the view volume, but setting the `VIEWPLACE_RECT` is a good method for displaying images. The model is responsible for any transformations like rotating, scaling, or translating that are needed:

```
IDL> model = obj_new('IDLgrModel')
```

The objects must be connected into a hierarchy. This is done via the `add` method for the container objects like the view and model:

```
IDL> view->add, model
```

Next, create the graphics atom that represents the image and add it to the hierarchy as well:

```
IDL> image = obj_new('IDLgrImage', ali)
IDL> model->add, image
```

Note that the `IDLgrImage` object is smart enough to determine the interleave of the image. Finally, a destination must be created to render the visualization:

```
IDL> window = obj_new('IDLgrWindow', dimensions=[256, 256])
```

There are several destination classes for rendering output to graphics windows, memory buffers, VRML files, the system clipboard, or directly to a printer. The last step is to tell the destination to draw the view:

```
IDL> window->draw, view
```

This should produce the following display:



Display of example data file *people.jpg* using object graphics.

All IDL's provided classes, not just the object graphics classes, use a property interface that coordinates the keywords to the `init`, `getProperty`, and `setProperty` methods. For example, the image's `INTERLEAVE` property can be retrieved after the image is created:

```
IDL> image->getProperty, interleave=interleave
IDL> print, interleave
0
```

Properties can be set as well, but remember that the window must be redrawn to see the changes:

```
IDL> image->setProperty, channel='ff0000'x
IDL> window->draw, view
```

Properties can also be set when the object is created with `OBJ_NEW`, i.e., when calling the `init` method. Some properties cannot be used in all the property handling methods. The documentation for the class has a “Properties” page that describes each property and indicates in which methods it can be used.

All the objects should be destroyed using `OBJ_DESTROY` when finished with them. Because the objects in the hierarchy are connected, freeing the top-level container, the view in this case, will free all the objects in the hierarchy:

```
IDL> obj_destroy, [view, window]
```

The window object is not part of the hierarchy so it must be freed as well, but it is freed automatically if the window is closed normally, i.e., by clicking the “X” in the title bar of the window. It can be convenient to use the `GRAPHICS_TREE` property of a window to associate a particular graphics hierarchy with the window. In that case, freeing the window will cause the graphics hierarchy to be freed as well.

9.3. The theory

Classes in the object graphics system can be broken down into five types: containers, graphics atoms, destinations, attributes, and helpers. Only containers and graphics atoms combine to form the object graphics hierarchy. Destinations draw the object graphics hierarchy to output destinations like a graphics window, the system clipboard, a memory buffer, or a graphics file. Attribute objects, like font or symbol objects, describe some aspect of graphics atoms. Helpers do various calculations related to object graphics.

Containers all inherit from *IDL_Container* and, therefore, share several methods to deal with their children. The *add* method builds the hierarchy:

```
container->add, objects [, position=index]
```

The other methods are more useful for general containers, but occasionally used in graphics hierarchies. The *get* method can retrieve children of a container:

```
result = container->get([, /all [, isa=classname(s)] | [, position=index] [, count=variable])
```

There are a few methods to query a container:

```
result = container->count()
result = container->isContained(object [, position=variable])
```

There are also some methods that can reorder or remove the children of a container:

```
container->move, source, destination
container->remove [, child_object] [, position=index] [, /all]
```

All *IDLgr* container classes also have a *getByName* method that finds children in the object graphics hierarchy by combining the objects’ *NAME* property with separating */s* to form a string similar to a file path:

```
child = container->getByName(name)
```

Each container also has a purpose beyond simply containing other objects. The container classes and their uses are described in Table 9.1, “Object graphics container classes” [p. 245].

Table 9.1. Object graphics container classes

Class	Description
<i>IDLgrScene</i>	An optional top-level container, a scene represents the entire visualization. Scenes contain views or viewgroups.
<i>IDLgrViewGroup</i>	A viewgroup is similar to a scene. Viewgroups can contain views or objects outside of the object graphics hierarchy. These other objects are added to a viewgroup for automatic destruction when the object graphics hierarchy is freed. Viewgroups are not required, but are convenient to free related objects automatically.
<i>IDLgrView</i>	A view represents a viewpoint into the 3-dimensional scene of the graphics. For example, it has properties for setting the size of the view volume (the rendered portion of the scene), the location

Class	Description
	of the eye, and the projection used by the view. It is necessary to display a graphics atom. A view contains models.
<i>IDLgrModel</i>	A model is responsible for transforming (rotating, translating, or scaling) the child objects it contains. Models contain other models or graphics atoms. They are required to display graphics atoms.

Graphics atoms represent something that will be displayed in the scene. Without them, only the background color will appear. They hold the data and its visualization properties. The graphics atom classes are listed below.

Table 9.2. Object graphics atom classes

Class	Description
<i>IDLgrAxis</i>	Represents a single axis with its labels, tickmarks, and title.
<i>IDLgrContour</i>	Represents a contour plot.
<i>IDLgrImage</i>	Represents an image (2- or 3-dimensional array with or without an alpha channel with any interleave).
<i>IDLgrLight</i>	Represents a light source: ambient, directional, positional, or spotlight.
<i>IDLgrPlot</i>	Represents a line or scatter plot.
<i>IDLgrPolygon</i>	Represents a set of polygons.
<i>IDLgrPolyline</i>	Represents a set of line segments.
<i>IDLgrROI</i>	Represents a region of interest.
<i>IDLgrROIGroup</i>	Represents a group of regions of interest.
<i>IDLgrSurface</i>	Represents a surface plot of various styles: points, wire frame, solid, ruled, or lego.
<i>IDLgrVolume</i>	Represents a 3-dimensional volume of data.

A light graphics atom is slightly different than other atoms in that it does not directly appear, but changes the appearance of the other atoms displayed by highlighting and shading the contours of 3-dimensional objects in the scene.

Several atoms can be combined to form reusable components called *composite classes*, like *IDLgrLegend* and *IDLgrColorbar* provided in the IDL library. These classes contain a model with several children that together perform some higher-level function. An example of creating a composite class is shown in Section 9.10, “Composite graphics classes” [p. 282].

Table 9.3. Object graphics composite classes

Class	Description
<i>IDLgrColorbar</i>	Represents a colorbar with an axis and title.
<i>IDLgrLegend</i>	Represents a legend showing a correspondence between glyphs and names.

The attribute classes specify a reusable attribute of graphics atom classes. An object of one of these classes can be used as an attribute of several atoms. Attribute classes are not part of the graphics hierarchy, so they are often added to a viewgroup to be cleaned up automatically when the group is destroyed.

Table 9.4. Object graphics attribute classes

Class	Description
<i>IDLgrFont</i>	Attribute of <i>IDLgrText</i> objects representing a typeface, style, weight, and size.

Class	Description
<i>IDLgrPalette</i>	Attribute of several atom classes representing a color table.
<i>IDLgrPattern</i>	Attribute of <i>IDLgrPolygon</i> objects describing the fill pattern.
<i>IDLgrSymbol</i>	Attribute of <i>IDLgrPlot</i> and <i>IDLgrPolyline</i> objects describing a plotting symbol.

Some graphics atom classes, like *IDLgrText* and *IDLgrImage*, also act as attributes. For example, an *IDLgrText* object can be added directly to the object graphics hierarchy or it can be the value of the *TITLE* property of an *IDLgrAxis* object. In the second case, the text object is acting as an attribute: it is not part of the hierarchy, it can be used for other text values, and it will not be cleaned up automatically with the hierarchy unless it was also added to a viewgroup.

Destination classes represent output formats; they are the analog of graphics devices in direct graphics. Each destination has a *draw* method used to render a graphics hierarchy:

```
dest->draw, scene
```

where *scene* is the top-level container in an object graphics hierarchy: a scene, viewgroup, or view.

Table 9.5. Object graphics destination classes

Class	Description
<i>IDLgrBuffer</i>	Memory buffer that can be read as an image after the graphics are drawn to it; this is the object graphics analog to a direct graphics pixmap.
<i>IDLgrClipboard</i>	The system clipboard whose contents can then be pasted into other applications. The clipboard is also used to output vector graphics.
<i>IDLgrPDF</i> ^{8.0}	Destination used to send output to a PDF file.
<i>IDLgrPrinter</i>	Destination used to send output directly to a printer.
<i>IDLgrVRML</i>	Destination to send output to a VRML file.
<i>IDLgrWindow</i>	Standard graphics window.

^{8.0}the PDF destination class was added in IDL 8.0

Destinations have a *GRAPHICS_TREE* property that can be used to connect a graphics hierarchy to a destination. The destination will then free the graphics hierarchy when it is destroyed and the top-level container of the hierarchy does not need to be passed to the destination's *draw* method.

Helper classes perform a variety of tasks related to object graphics.

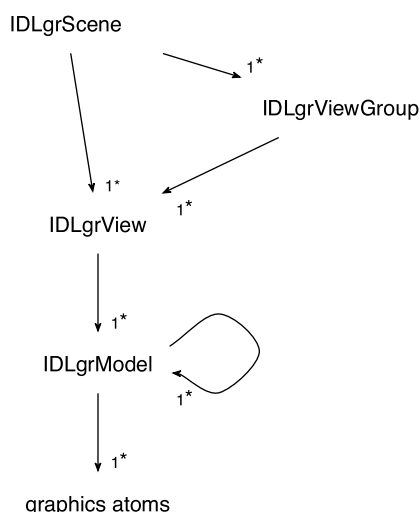
Table 9.6. Object graphics helper classes

Class	Description
<i>IDLgrMPEG</i>	Creates MPEG movies.
<i>IDLgrFilterChain</i> ^{6.4}	Container class containing shader objects to be applied to an image object.
<i>IDLgrShader</i> ^{6.4}	Provides access to doing image processing computations on the graphics processing unit (GPU).
<i>IDLgrShaderBytscl</i> ^{6.4}	Subclass of <i>IDLgrShader</i> that applies a BYTSCL operator to an image object.
<i>IDLgrShaderConvolve</i> ^{6.4}	Subclass of <i>IDLgrShader</i> that applies 3 by 3 convolution operators to an image object.
<i>IDLgrTessellator</i>	A tessellator object decomposes a polygon into a set of triangles.
<i>Trackball</i>	Trackball objects transform mouse movements to transformation matrices; a useful utility for rotating the contents of models in widget programs.

^{6.4}the shader classes were added in IDL 6.4

The structure of the object graphics hierarchy is shown in the diagram below. It must be rooted with a scene, view, or viewgroup. Several views can be added if the hierarchy starts with a scene or viewgroup. Each view can be positioned independently in the display. A model must be present to display a graphics atom. Models can be nested arbitrarily deep to allow for specific branches of the hierarchy to be transformed, i.e., rotated, scaled, or translated without affecting the rest of the hierarchy.

Figure 9.1. General object graphics hierarchy



The general structure for an object graphics hierarchy can be arbitrarily complicated. The hierarchy can be rooted at a scene, viewgroup, or view. The “1*” labels on the connections indicate one or more connections can be made.

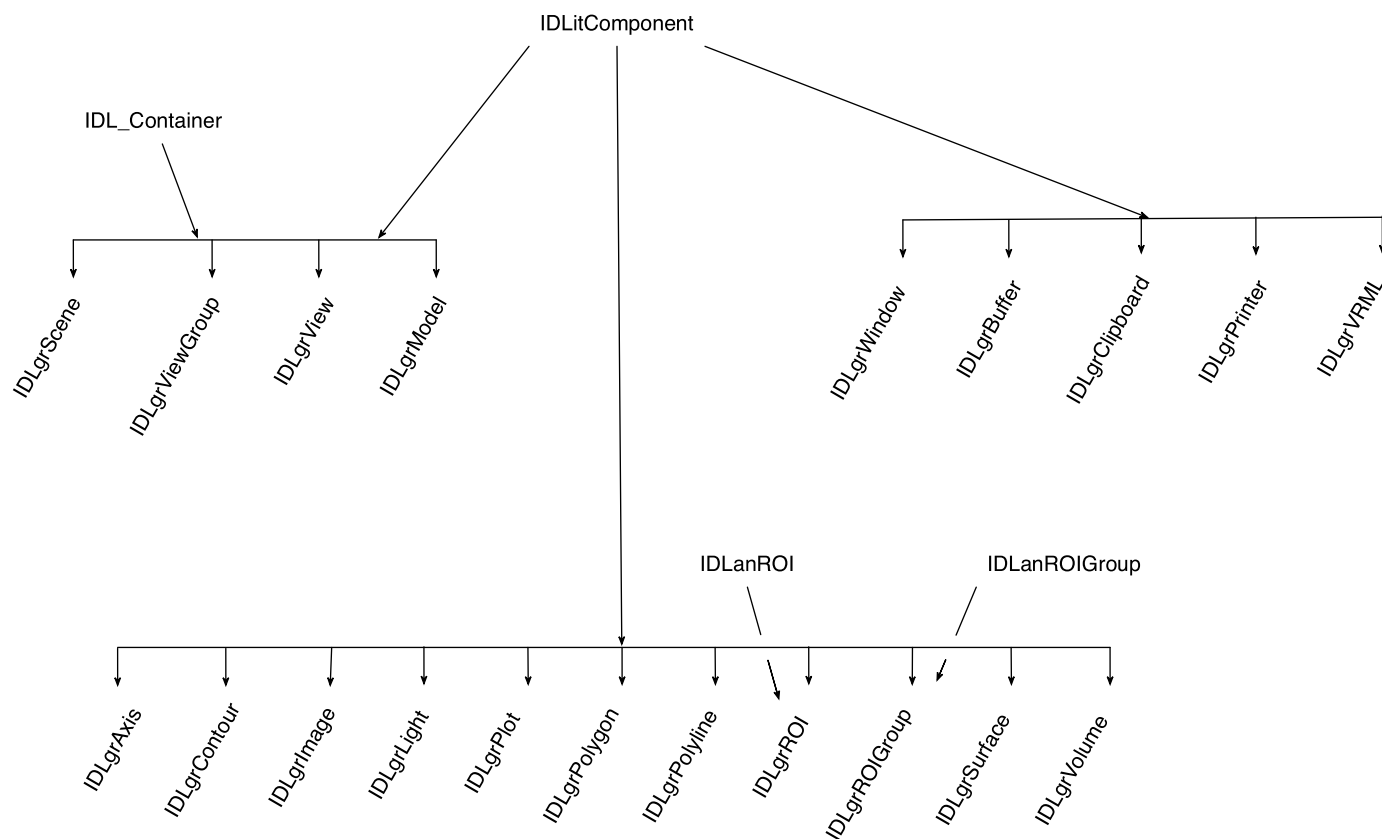
The standard way to set and get attributes of an object in IDL is to use the object’s *setProperty* and *getProperty* methods with the attribute (i.e., property) name as the keyword name. The *IDLgr* classes, as well as nearly all of the IDL’s class library, make use of the following interface to manipulate properties:

```
ogr->getProperty, property_name=pvalue
ogr->setProperty, property_name=pvalue
```

Properties can also be set on the creation of an object via the *init* method. The documentation for each class has a “Properties” page with a description of each property and which methods it can be used with (some properties cannot be set, others cannot be retrieved).

Classes in the object graphics library inherit from *IDLitComponent*.

Figure 9.2. Class diagram for objects in graphics hierarchy



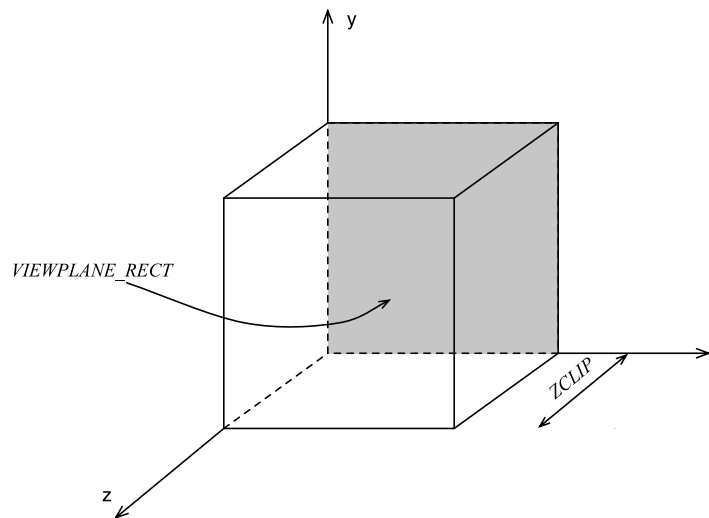
Undocumented parent classes are not shown in this diagram.

9.4. A 3-dimensional example

Direct graphics and object graphics can produce similar output for most 2-dimensional graphics like images and line plots, but object graphics has several advantages for 3-dimensional output. Object graphics are inherently 3-dimensional —making composition of scenes with proper perspective much easier. Other, more advanced, features are also available, like texture mapping, transparency, and cutting planes.

One important task in object graphics is to match up the display coordinates of the view volume with the data coordinates.

In this example, instead of setting the view's `VIEWPLANE_RECT` keyword (along with the `ZCLIP` keyword for 3-dimensional scenes), the graphics atom's `[XYZ]COORD_CONV` will be used to scale the data coordinates into the default *view volume* (-1 to 1 in each direction).

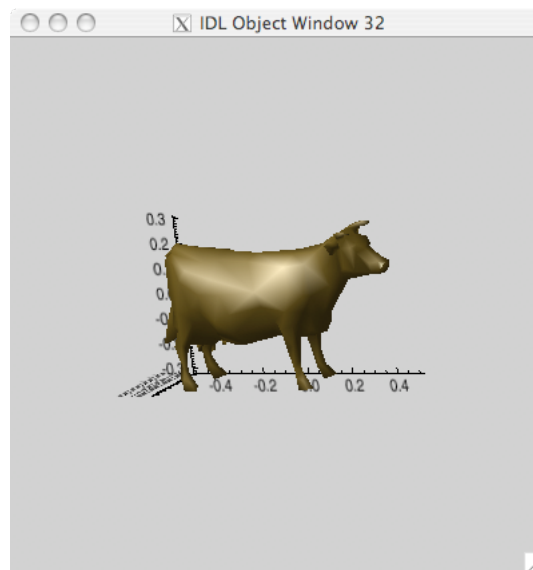


The view volume for an object graphics scene is defined by the `VIEWPLANE_RECT` and `ZCLIP` keywords to the `IDLgrView`. By default, the view volume extends from -1 to 1 in each direction.

The example routine shown below, `MG_SHOW_POLYGON`, displays a set of vertices and a connectivity list as a polygon. The connectivity list specifies how the vertices are connected to form polygons in the same manner as direct graphics meshes, see Section 6.12, “Polygonal meshes” [p. 170] for a description. If no vertices are passed in, the `cow10.sav` data set in the IDL distribution is used. To run the example with the cow polygons, type:

```
IDL> mg_show_polygon
```

This produces the following visualization:



Display of example data file `cow10.sav` using the object graphics program, `MG_SHOW_POLYGON`.

To display arbitrary polygonal meshes, vertices are passed in a three separate vectors *x*, *y*, and *z* and the connectivity list is passed in through the *POLYGONS* keyword. The cow data set is read in if no arguments are present. The *RENDERER* keyword specifies hardware (0, the default) or software (1) rendering.

```
pro mg_show_polygon, x, y, z, polygons=polylist, renderer=renderer
  compile_opt strictarr

  ; default data set is a cow
  if (n_params() ne 3) then begin
    ; restore cow.sav
    filename = filepath('cow10.sav', subdir=['examples', 'data'])
    restore, filename=filename
  endif
```

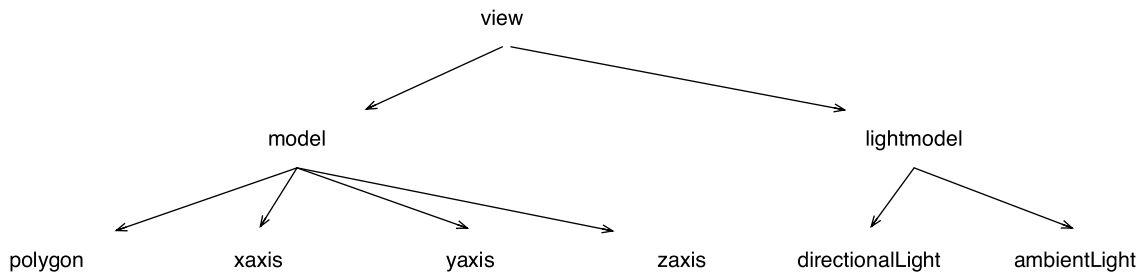
Next, the basic object graphics hierarchy is created, rooted at a view:

```
; view is the top-level object in this hierarchy
view = obj_new('IDLgrView', color=[200, 200, 200])

; the model controls the transformation matrix (rotation, translation, scaling)
model = obj_new('IDLgrModel')
view->add, model

; the polygon itself
polygon = obj_new('IDLgrPolygon', x, y, z, $ ; the vertices
                  polygons=polylist, $      ; how the vertices form polygons
                  color=[100, 80, 25], $
                  shading=1)                ; Gouraud shading
model->add, polygon
```

We have created the left most side of the final hierarchy, which will look like the following:



A separate model is created for the lights so that the polygon can be rotated independently from the lights. Two lights are created and added to *lightModel*: a directional light (type=2) and an ambient light (type=0):

```
; the lights are in a separate model so they don't rotate with the polygon
lightmodel = obj_new('IDLgrModel')
view->add, lightmodel

directionalLight = obj_new('IDLgrLight', type=2, location=[-1, 1, 1], intensity=0.7)
lightmodel->add, directionalLight
ambientLight = obj_new('IDLgrLight', type=0, intensity=0.3)
lightmodel->add, ambientLight
```

The following code scales the polygon into the view volume. All dimensions are scaled equally to preserve the aspect ratio, so the range of each dimension is found so that the largest size can be used to calculate the coordinate conversion function for all dimensions.

```
; The next few lines of code is the tricky part to scale the polygon
; correctly into the view volume.
polygon->getProperty, xrange=xr, yrange=yr, zrange=zr

; find the dimension the polygon has the largest range in and use that to
; scale the all the dimensions; alternatively, each dimension could have its
; own scaling function so each dimension takes up the same amount of display
; space no matter what its data range
ranges = [[xr], [yr], [zr]]
m = max(ranges[1, *] - ranges[0, *], maxRangeRow)

; find the coordinate conversion function that will scale the given range
; into [0, 1]
cc = norm_coord(ranges[, maxRangeRow])

; now translate it over, so it scales the given range into [-0.5, 0.5]
cc[0] -= 0.5

; tell the polygon what the coordinate conversion function is (notice all
; dimensions use the same one, i.e., isotropic scaling)
polygon->setProperty, xcoord_conv=cc, ycoord_conv=cc, zcoord_conv=cc
```

The NORM_COORD function works well to create a coordinate conversion function as long as the scaled item is intended to take up one unit (i.e., normalized), in this case -0.5 to 0.5. Use MG_LINEAR_FUNCTION when other sizes are needed; see following examples for usage.

Next, *x*-, *y*-, and *z*-axes are created with the same coordinate conversion function as the polygon so they will be scaled in the same manner:

```
; create axes with the same scaling as the cow
xaxis = obj_new('IDLgrAxis', 0, range=xr, /exact, $
               location=[xr[0], yr[0], zr[0]], $
               xcoord_conv=cc, ycoord_conv=cc, zcoord_conv=cc, $
               ticklen=0.025)
model->add, xaxis
yaxis = obj_new('IDLgrAxis', 1, range=yr, /exact, $
               location=[xr[0], yr[0], zr[0]], $
               xcoord_conv=cc, ycoord_conv=cc, zcoord_conv=cc, $
               ticklen=0.025)
model->add, yaxis
zaxis = obj_new('IDLgrAxis', 2, range=zr, /exact, $
               location=[xr[0], yr[0], zr[0]], $
               xcoord_conv=cc, ycoord_conv=cc, zcoord_conv=cc, $
               ticklen=0.025)
model->add, zaxis
```

Often it is necessary to rotate a 3-dimensional object to get a better perspective on it. The default view of the cow would be directly at the right side of the cow. It is rotated slightly in this example to show the top and front of the cow also:

```
; rotate to get a nicer original orientation
model->rotate, [1, 0, 0], 15
model->rotate, [0, 1, 0], -30
```

Finally, an *IDLgrWindow* object is created to display the polygon:

```

window = obj_new('IDLgrWindow', dimensions=[400, 400], renderer=renderer)
window->draw, view
obj_destroy, view
end

```

This produces a static visualization of the polygon, so the view can be destroyed as soon as the window has drawn it. In interactive visualizations, the view should be destroyed only when it will no longer be updated.

The XOBJVIEW procedure can be useful to easily produce an interactive visualization of object graphics atoms. To show the above display using XOBJVIEW, first load the cow data set:

```

IDL> filename = filepath('cow10.sav', subdir=['examples', 'data'])
IDL> restore, filename=filename, /verbose

```

Next, create just the polygon object:

```

IDL> polygon = obj_new('IDLgrPolygon', x, y, z, polygons=polylist, color=[100, 80, 25], shading=1)

```

Finally, pass the polygon to XOBJVIEW:

```

IDL> xobjview, polygon

```

The XOBJVIEW procedure can accept either a graphics atom or a model containing atoms or other models. Be sure to clean up the polygon object when finished with it:

```

IDL> obj_destroy, polygon

```

9.5. The transformation matrix

A *transformation matrix* is a 4 by 4 matrix representation of the orientation of graphic atoms and of operations such as rotations, translations, and scalings to be performed on them. In object graphics, it is contained in the *TRANSFORM* property of an *IDLgrModel* and is applied to all the objects under the model in the graphics hierarchy. An atom may have several transformation matrices applied to it, not just the transform of its immediate parent, because of the allowance for models to contain other models. To get the cumulative transformation matrix applied to an object, use the object's *getCTM* method.

A transformation matrix T is applied to a point x by the equation

```
newX = x # T
```

Multiple transformations can be composed by multiplying them together first before applying to the vector. To apply transform T_0 , followed by transform T_1 do

```
newX = (x # T0) # T1
```

or

```
new = x # (T0 # T1)
```

For an example, let's look at the transformation matrix held by a model as several transformations are applied to it. First, create a model:

```

IDL> model = obj_new('IDLgrModel')

```

Its transformation matrix is held in the *TRANSFORM* property:

```

IDL> model->getProperty, transform=transform
IDL> print, transform
      1.0000000      0.0000000      0.0000000      0.0000000

```

```
0.0000000    1.0000000    0.0000000    0.0000000
0.0000000    0.0000000    1.0000000    0.0000000
0.0000000    0.0000000    0.0000000    1.0000000
```

The default transformation matrix is the identity matrix, indicating no transformation. Now, let's translate the model and see the effect on the transformation matrix:

```
IDL> model->translate, 2, 3, 4
IDL> model->getProperty, transform=transform
IDL> print, transform
1.0000000    0.0000000    0.0000000    2.0000000
0.0000000    1.0000000    0.0000000    3.0000000
0.0000000    0.0000000    1.0000000    4.0000000
0.0000000    0.0000000    0.0000000    1.0000000
```

Formulas for the general relationships will be given, but it is clear from the above that the first three elements of the last column hold the translation of the transform. Now, scale the model:

```
IDL> model->scale, 5, 6, 7
IDL> model->getProperty, transform=transform
IDL> print, transform
5.0000000    0.0000000    0.0000000    10.000000
0.0000000    6.0000000    0.0000000    18.000000
0.0000000    0.0000000    7.0000000    28.000000
0.0000000    0.0000000    0.0000000    1.0000000
```

This multiplies each row by the appropriate scaling factor (again, formulas will be given later). Finally, a rotation about the x -axis is done:

```
IDL> model->rotate, [1, 0, 0], 90
IDL> model->getProperty, transform=transform
IDL> print, transform
5.0000000    0.0000000    0.0000000    10.000000
0.0000000   -2.2971412e-15   -7.0000000   -28.000000
0.0000000    6.0000000   -2.6799981e-15    18.000000
0.0000000    0.0000000    0.0000000    1.0000000
```

This relationship is more complicated and affects the 3 by 3 matrix in the upper-right of the transformation matrix. The general form is much more complicated than the translation and scaling formulas. When finished with the model, destroy it:

```
IDL> obj_destroy, model
```

For an interactive example of displaying the transformation matrix for a 3-dimensional object you can rotate, translate, and scale, try running MG_TRANSFORM_DEMO:

```
IDL> mg_transform_demo
```

This can be a useful way for getting a feel for how the transformation matrix changes as a 3-dimensional object is controlled.

For a more mathematical specification of transformations, a translation by (D_x, D_y, D_z) is represented by the following transformation matrix:

```
1    0    0    Dx
0    1    0    Dy
0    0    1    Dz
```

$$\begin{matrix} 0 & 0 & 0 & 1 \end{matrix}$$

Scaling by (S_x, S_y, S_z) , each dimension can be scaled by a different factor, is represented by the following transformation matrix:

$$\begin{matrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Finally, rotating by an angle t , in radians, about an axis specified by the unit vector (x, y, z) is represented by the following matrix:

$$\begin{matrix} (1 - \cos t) x^2 + \cos t & (1 - \cos t) xy - z \sin t & (1 - \cos t) xz + y \sin t & 0 \\ (1 - \cos t) xy + z \sin t & (1 - \cos t) y^2 + \cos t & (1 - \cos t) yz - x \sin t & 0 \\ (1 - \cos t) xz - y \sin t & (1 - \cos t) yz + x \sin t & (1 - \cos t) z^2 + \cos t & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

This is the calculation performed by the *IDLgrModel::rotate* method.

The above rotation calculation is always with respect to a vector, i.e., an axis of rotation passing through the origin. It is smart to choose an origin which is the center of rotations that will be performed on the objects in the display, i.e., usually the middle of the display works well and is a good reason to keep the default view volume coordinates of -1 to 1 in each dimension. But if two objects need to be rotated about differing points in the same object graphics scene, then it is impossible to choose an origin that satisfies the requirement for both objects. In this case, the following simple technique is used:

1. Translate the given model from the point of rotation to the origin.
2. Perform the rotation.
3. Translate the model back from the origin to the original location.

For example, this technique would be needed to create a visualization of the solar system with planets rotating both around the sun and their own axes. This trick is also useful when scaling.

The `MG_SHOW_AXES` procedure is one example of applying facts about the transformation matrix to an IDL application. It rotates the label on the axes to appear in the correct direction as the axes are rotated. Try the example with

```
IDL> mg_show_axes
```

To run the example without switching the labels, try

```
IDL> mg_show_axes, /no_switch
```

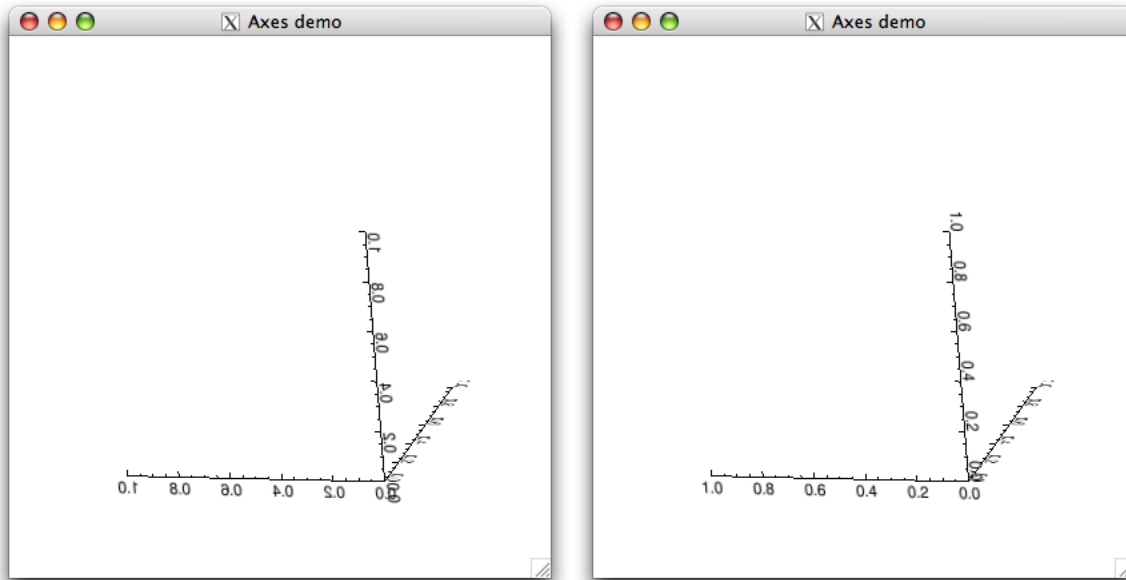
The key fact used in this example, is that the element at $[2, 2]$ indicates whether the axes labels are turned towards or away from the viewer. This is a rotation about $(0, 1, 0)$, so

$$(1 - \cos t) z^2 + \cos t < 0$$

becomes

$$\cos t < 0$$

which happens exactly when the angle t is between 90 and 270 degrees, i.e., when the axes are turned away.



Running the MG_SHOW_AXES demo program without switching the orientation of the text (on the left) and with switching (the default, on the right).

9.6. Properties

Properties are attributes of an object that are accessed through keywords to the *getProperty* and *setProperty* methods as well as when instantiating the object, i.e., using *OBJ_NEW* to create the object. For example, an *IDLgrSurface* object has a *COLOR* property to control the color of the surface. The color can be set when the surface is created or later via the *setProperty* method. The current color can be obtained with the *getProperty* method.

The online help contains a “Properties” page for each class in the IDL library. This page lists details about each property including a table like the one below:

Property Type	Floating-point vector		
Name String	not displayed		
Get: Yes	Set: Yes	Init: Yes	Registered: No

Some properties can be get, set, and used when instantiating the object.

The “Get”, “Set”, and “Init” entries of the above table indicate whether the property can be used with *getProperty*, *setProperty*, and *init*, respectively. Most properties can be accessed in all three methods, but some cannot:

Property Type	Floating-point vector		
Name String	<i>not displayed</i>		
Get: Yes	Set: No	Init: No	Registered: No

Some properties are limited to where they can be accessed.

The above property can be retrieved with `getProperty`, but cannot be set with `setProperty` or `init`.

The “Registered” entry specifies whether the property will appear in a property sheet with the object set as the value. The `REGISTER_PROPERTIES` property of the object must be set for the properties to be registered. Properties can be registered with the `IDLitComponent::registerProperty` method (all the classes of objects that appear in the object graphics hierarchy inherit from `IDLitComponent`).

Some common properties of graphics atoms are listed in Table 9.7, “Common *IDLgr* atom class properties” [p. 257]. Most atoms, except for *IDLgrLight*, have all the listed properties plus other properties specific to the particular atom type.

Table 9.7. Common *IDLgr* atom class properties

Class	Description
<i>ALL</i>	Read-only property holding a structure with each property of the object in a field.
<i>ALPHA_CHANNEL</i>	Floating point value for the transparency of the atom where 0.0 is completely transparent and 1.0 is completely opaque (default).
<i>CLIP_PLANES</i>	Coefficients of clipping planes applied to the atom. The four coefficients for a plane [a, b, c, d] clip the portion of the atom in $ax + by + cz + d > 0$
<i>COLOR</i>	Color of the atom as an RGB triplet.
<i>HIDE</i>	Boolean value determining whether to show the object and its children.
<i>PARENT</i>	Read-only value containing the parent object in the graphics hierarchy.
<i>REGISTER_PROPERTIES</i>	A boolean value that registers the properties of this object that are marked “Registered” in the online help. Registered properties appear in a property sheet with this object as the value. This property can only be accessed on instantiation.
<i>SHADER</i>	An <i>IDLgrShader</i> object which will be used to render the atom if a compatible graphics card is present.
<i>[XYZ]COORD_CONV</i>	Two-element vectors specifying linear functions to map data coordinates into the display coordinates of the view volume.
<i>[XYZ]RANGE</i>	Read-only property specifying limits of the data coordinates in each dimension.

9.7. Sample visualizations

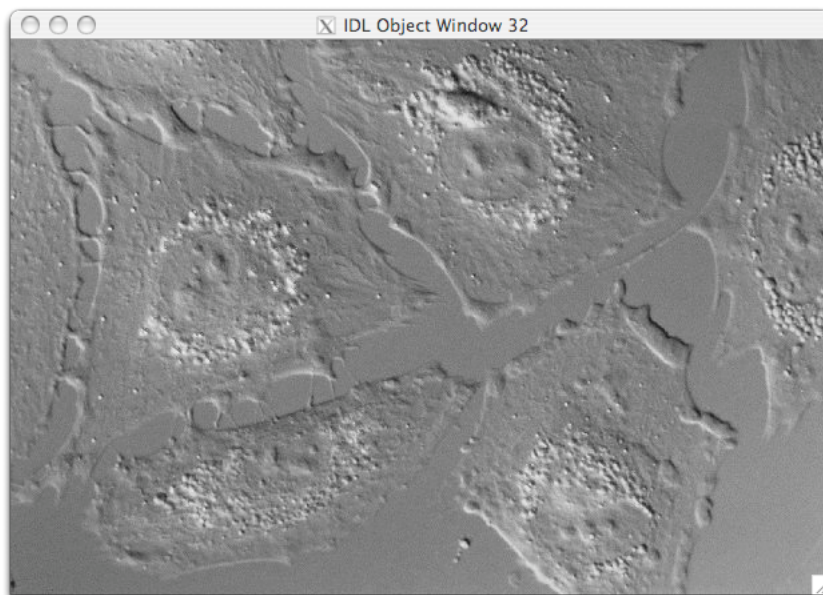
One of the easiest ways to accomplish a new task is to modify existing code that accomplishes a similar task. In the spirit of that concept, this section presents examples of creating simple visualizations of the basic types: image display, line plots, surface plots, contour plots, volume visualizations, polygonal mesh visualizations, and maps. All the source code for these visualizations is provided and discussed to give a starting point for new visualizations.

The routine `MG_OGIMAGE_EXAMPLE` will display the given array as an image. For example, load an image and display it:

```
IDL> endoFilename = filepath('endocell.jpg', subdir=['examples', 'data'])
IDL> endo = read_image(endoFilename)
```

```
IDL> mg_ogimage_example, endo
```

This produces the following display:

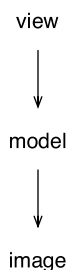


Display of example data file *endocell.jpg* using the object graphics program *MG_OGIMAGE_EXAMPLE*.

The easiest way to set up the coordinate system for images is to use the *VIEWPLANE_RECT* property of the *IDLgrView* class to define a coordinate system that corresponds to the pixels of the image (like device coordinates in direct graphics). The dimensions of the image are retrieved and the coordinate system is defined in the following lines of *MG_OGIMAGE_EXAMPLE*:

```
image->getProperty, dimensions=dims
view->setProperty, viewplane_rect=[0, 0, dims - 1L]
```

The graphics hierarchy for this example is the minimal tree necessary to view any atom:



The complete code for *MG_OGIMAGE_EXAMPLE* is quite short. The trick here is to grab the dimensions of the image from the *DIMENSIONS* property of the *IDLgrImage* and use that to set the *IDLgrView*'s *VIEWPLANE_RECT* property. This allows an image of any interleave type to be input without complicated code to compute the dimensions.

```
pro mg_ogimage_example, im, _extra=e
    compile_opt strictarr
```



```

view = obj_new('IDLgrView')

model = obj_new('IDLgrModel')
view->add, model

image = obj_new('IDLgrImage', im, _extra=e)
model->add, image
image->getProperty, dimensions=dims

view->setProperty, viewplane_rect=[0, 0, dims - 1L]

window = obj_new('IDLgrWindow', dimensions=dims, graphics_tree=view, _extra=e)
window->draw
end

```

Displaying images is often quite simple, but see Section 9.14, “Widgets and object graphics: tiled imagery” [p. 293] for a more complicated example using tiling, i.e., using only the data necessary to display the part of the image currently visible at the required zoom level. Also, see some of the plotting examples in this section for how to add axes to the image if required.

The next example will display a line plot. A dampened sine curve will be used for the data. Read the data with

```

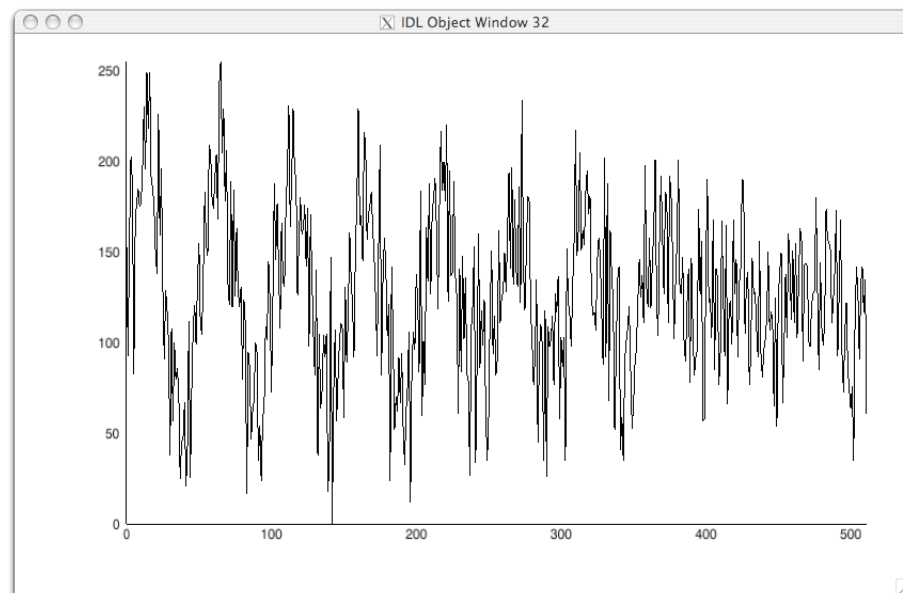
IDL> openr, lun, filepath('damp_sn.dat', subdir=['examples', 'data']), /get_lun
IDL> dsin = bytarr(512)
IDL> readu, lun, dsin
IDL> free_lun, lun

```

Both x and y vectors are needed for calling the example, so create a simple index array for the x variable:

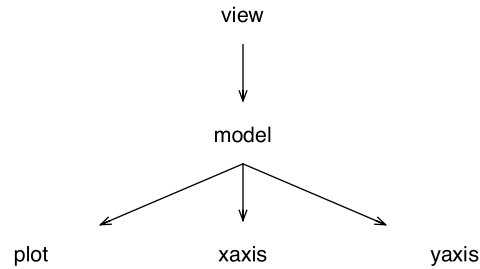
```
IDL> mg_ogplot_example, findgen(512), dsin
```

This should result in the following output:



Display of example data file damp_sn.dat using the object graphics program MG_OG_PLOT_EXAMPLE.

The object graphics hierarchy for this example is a bit more complicated by the addition of axes:



In this example, the `[XY]COORD_CONV` properties are used to scale the data coordinates into the view volume. The `MG_LINEAR_FUNCTION` computes the coefficients of the linear equation to scale from the input range to the given output range, i.e., `[-0.75, 0.90]` in this example.

```

pro mg_ogplot_example, x, y, _extra=e
  compile_opt strictarr

  view = obj_new('IDLgrView')

  model = obj_new('IDLgrModel')
  view->add, model

  plot = obj_new('IDLgrPlot', x, y, _extra=e)
  model->add, plot

  plot->getProperty, xrange=xr, yrange=yr
  xc = mg_linear_function(xr, [-0.75, 0.90])
  yc = mg_linear_function(yr, [-0.75, 0.90])
  plot->setProperty, xcoord_conv=xc, ycoord_conv=yc

  xaxis = obj_new('IDLgrAxis', direction=0, range=xr, /exact, _extra=e)
  model->add, xaxis

  yaxis = obj_new('IDLgrAxis', direction=1, range=yr, /exact, _extra=e)
  model->add, yaxis

  xaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc
  yaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc

  window = obj_new('IDLgrWindow', graphics_tree=view, _extra=e)
  window->draw
end
  
```

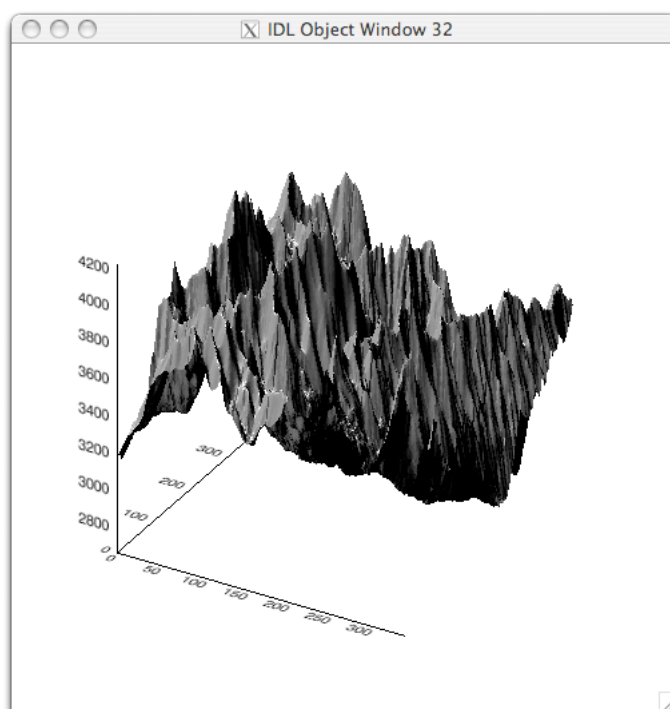
It is important to remember to set the `[XY]COORD_CONV` property values for the axes to the same values used for the plot itself.

The next example creates a 3-dimensional surface visualization. To run the example program, try

```

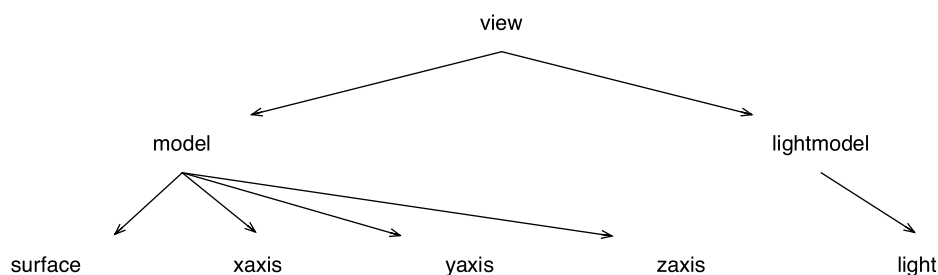
IDL> restore, filename=filepath('marbells.dat', subdir=['examples', 'data'])
IDL> mg_ogsurface_example, elev, color=[200, 200, 200], style=2
  
```

This should produce a display like shown below:



Display of Maroon Bells elevation data from `marbells.dat` in the example data of the IDL distribution using the example program `MG_OGSURFACE_EXAMPLE`. A real-world display would need to account for the differing horizontal and vertical scales, but our data set does not provide the necessary data to determine the vertical exaggeration.

The object graphics hierarchy puts a surface object and the three axes in the *model* `IDLgrModel` with the same coordinate conversion functions, while putting a directional light in the *lightmodel* model.



The code for the surface example is given below. Keywords to the example routine are passed along to the creation of the `IDLgrSurface`. The `RENDERER` indicates hardware (0, the default) or software (1) rendering.

```

pro mg_ogsurface_example, z, renderer=renderer, _extra=e
  compile_opt strictarr

  view = obj_new('IDLgrView')

  model = obj_new('IDLgrModel')
  
```

```

view->add, model

surface = obj_new('IDLgrSurface', z, _extra=e)
model->add, surface

lightModel = obj_new('IDLgrModel')
view->add, lightModel

light = obj_new('IDLgrLight', type=2, location=[-1, 1, 1])
lightModel->add, light

surface->getProperty, xrange=xr, yrange=yr, zrange=zr
xc = mg_linear_function(xr, [-0.5, 0.5])
yc = mg_linear_function(yr, [-0.5, 0.5])
zc = mg_linear_function(zr, [-0.5, 0.5])
surface->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc

model->rotate, [1, 0, 0], -90
model->rotate, [0, 1, 0], -30
model->rotate, [1, 0, 0], 30

xaxis = obj_new('IDLgrAxis', direction=0, range=xr, /exact, $
                location=[xr[0], yr[0], zr[0]])
model->add, xaxis

yaxis = obj_new('IDLgrAxis', direction=1, range=yr, /exact, $
                location=[xr[0], yr[0], zr[0]])
model->add, yaxis

zaxis = obj_new('IDLgrAxis', direction=2, range=zr, /exact, $
                location=[xr[0], yr[0], zr[0]])
model->add, zaxis

xaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc
yaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc
zaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc

window = obj_new('IDLgrWindow', dimensions=[500, 500], graphics_tree=view, renderer=renderer)
window->draw
end

```

The *[XYZ]COORD_CONV* property values would be set to the same linear function to provide an isotropic display.

Contour plots in object graphics can be difficult. To read the data, use

```

IDL> convecFilename = filepath('convec.dat', subdir=['examples', 'data'])
IDL> convec = read_binary(convecFilename, data_type=1, data_dims=[248, 248])

```

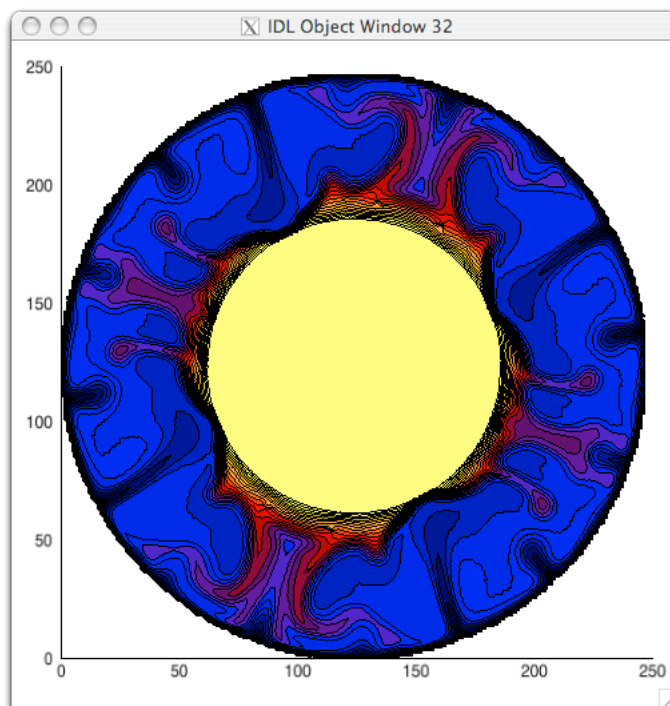
Run the example program with

```

IDL> mg_ogcontour_example, convec, n_levels=30

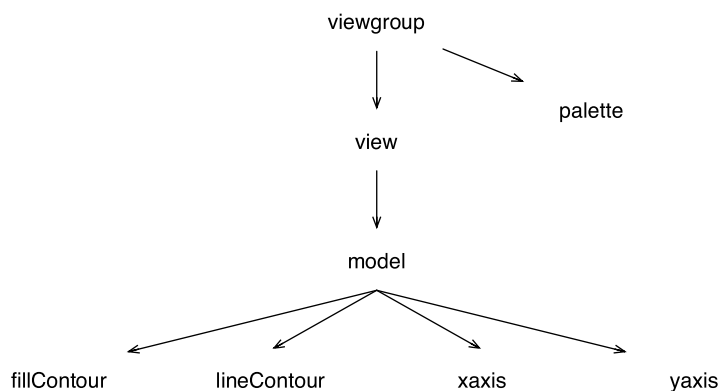
```

The contour plot should look like the following:



The *convec.dat* data set is displayed in a contour plot with the demo program *MG_OGCONTOUR_EXAMPLE*.

The object graphics hierarchy is rooted at a viewgroup so that a palette can be added to it for automatic clean up when the hierarchy is destroyed. A palette is considered an attribute object, used by objects to define a color table, but not part of the graphics hierarchy.



To make a standard 2-dimensional contour plot, set the *PLANAR* property of the *IDLgrContour* and specify the *z* level with the *GEOMZ* property. Without these keywords, the contour levels would be at the height of their respective *z* values in 3-dimensional space. Furthermore, in this display, contour outlines are plotted over filled contours, but this can cause stitching problems where there is a conflict over which contour plot is in front of the other. Use the *DEPTH_OFFSET*

keyword to indicate that the filled contour is in the back even though both contours have the same z value. Higher values of `DEPTH_OFFSET` are farther away from the viewer.

```
pro mg_ogcontour_example, z, n_levels=nlevels, _extra=e
  compile_opt strictarr

  _nlevels = n_elements(nlevels) eq 0L ? 20 : nlevels

  viewgroup = obj_new('IDLgrViewGroup')

  view = obj_new('IDLgrView')
  viewgroup->add, view

  model = obj_new('IDLgrModel')
  view->add, model

  fillContour = obj_new('IDLgrContour', z, $
    planar=1, geomz=0.0, $
    n_levels=_nlevels, $
    /fill, c_color=bytsc1(bindgen(_nlevels)), $
    depth_offset=1, $
    _extra=e)
  model->add, fillContour

  lineContour = obj_new('IDLgrContour', z, $
    planar=1, geomz=0.0, $
    n_levels=_nlevels, $
    _extra=e)
  model->add, lineContour

  fillContour->getProperty, xrange=xr, yrange=yr
  xc = mg_linear_function(xr, [-0.85, 0.9])
  yc = mg_linear_function(yr, [-0.85, 0.9])
  fillContour->setProperty, xcoord_conv=xc, ycoord_conv=yc
  lineContour->setProperty, xcoord_conv=xc, ycoord_conv=yc

  xaxis = obj_new('IDLgrAxis', direction=0, range=xr)
  model->add, xaxis

  yaxis = obj_new('IDLgrAxis', direction=1, range=yr)
  model->add, yaxis

  xaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc
  yaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc

  palette = obj_new('IDLgrPalette')
  viewgroup->add, palette
  palette->loadCT, 5

  fillContour->setProperty, palette=palette

  window = obj_new('IDLgrWindow', dimensions=[500, 500], graphics_tree=viewgroup, _extra=e)
  window->draw
end
```

Volume data can be some of the hardest (and slowest) to visualize. Let's visualize the black hole data found in the IDL distribution:

```
IDL> blackHoleFilename = filepath('cduskcD1400.sav', subdir=['examples', 'data'])
IDL> restore, filename=blackHoleFilename
```

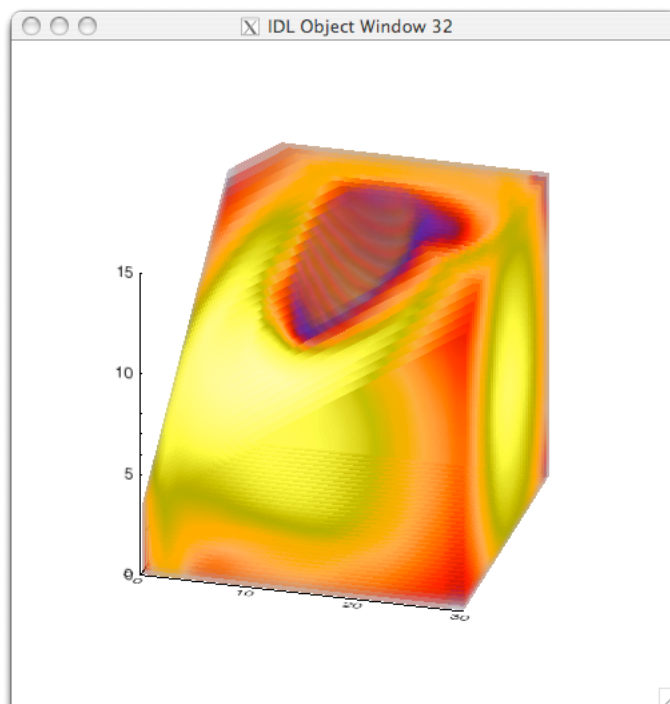
Run the example on the *density* variable, clipping away a corner of the volume to get a look at the interior:

```
IDL> mg_ogvolume_example, density, clip_planes=[-0.1, -0.1, 0.3, -1.0]
```

The *CLIP_PLANES* property is passed to the creation of the *IDLgrVolume*; it clips away the region described by

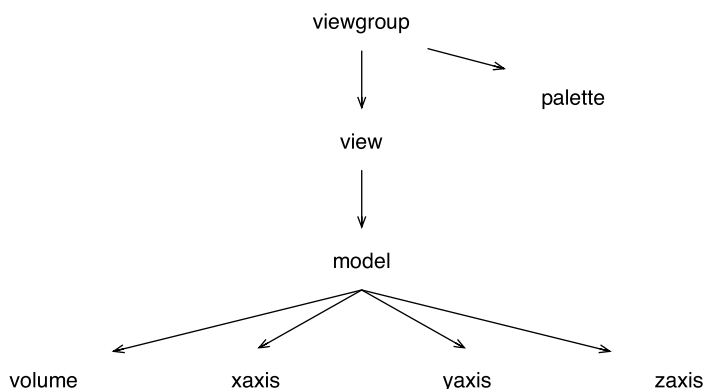
$$-0.1x - 0.1y + 0.3z - 1.0 > 0$$

The output should look like the following:



The black hole data stored in `cduskcD1400.sav` is shown as a volume with a clipping plane showing some of the interior using the `MG_OGVOLUME_EXAMPLE` procedure.

The graphics hierarchy is rooted at a viewgroup to provide a place for the palette so that it can be automatically freed. The volume and three axes are all in a single model making the hierarchy straightforward:



The *vol* is a 3-dimensional array to be displayed as a volume. The `MG_OGVOLUME_EXAMPLE` procedure accepts keywords for the `IDLgrVolume::init` or `IDLgrWindow::init` methods, which allows us to pass in the `CLIP_PLANES` keyword in the example call:

```

pro mg_ogvolume_example, vol, _extra=e
  compile_opt strictarr

  viewgroup = obj_new('IDLgrViewGroup')

  view = obj_new('IDLgrView', color=[0, 0, 0])
  viewgroup->add, view

  model = obj_new('IDLgrModel')
  view->add, model

  volume = obj_new('IDLgrVolume', bytscl(vol), interpolate=1, _extra=e)

  volume->getProperty, xrange=xr, yrange=yr, zrange=zr
  xc = mg_linear_function(xr, [-0.5, 0.5])
  yc = mg_linear_function(yr, [-0.5, 0.5])
  zc = mg_linear_function(zr, [-0.5, 0.5])
  volume->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc

  palette = obj_new('IDLgrPalette')
  viewgroup->add, palette
  palette->loadCT, 5
  palette->getProperty, red_values=r, green_values=g, blue_values=b

  volume->setProperty, rgb_table0=[[r], [g], [b]]
  model->rotate, [1, 0, 0], -90
  model->rotate, [0, 1, 0], -15
  model->rotate, [1, 0, 0], 25

  xaxis = obj_new('IDLgrAxis', direction=0, range=xr, color=[255, 255, 255])
  model->add, xaxis

  yaxis = obj_new('IDLgrAxis', direction=1, range=yr, color=[255, 255, 255])
  model->add, yaxis
  
```



```

zaxis = obj_new('IDLgrAxis', direction=2, range=zr, color=[255, 255, 255])
model->add, zaxis

model->add, volume

xaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc
yaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc
zaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc

window = obj_new('IDLgrWindow', dimensions=[500, 500], graphics_tree=viewgroup, _extra=e)
window->draw
end

```

Note that the *volume* was added to the *model* after the axes because generally items should be added starting from the back of the scene and proceeding to the front in case transparency is used (see Section 9.8, “Texture mapping and transparency” [p. 274] for details).

The *IDLgrVolume* class has several data, color table, and opacity properties that interact in different ways depending on the value of the *VOLUME_SELECT* property. The following table summarizes the different methods for computing the color and alpha channel values.

Table 9.8. Values of the *VOLUME_SELECT* property

Value	Description
0	<p>The values of <i>DATA0</i> are looked up in the color table in <i>RGB_TABLE0</i> to get the colors and in <i>OPACITY_TABLE0</i> to get the alpha channel values.</p> <pre> red[i, j, k] = RGB_TABLE0[DATA0[i, j, k], 0] green[i, j, k] = RGB_TABLE0[DATA0[i, j, k], 1] blue[i, j, k] = RGB_TABLE0[DATA0[i, j, k], 2] alpha[i, j, k] = OPACITY_TABLE0[DATA0[i, j, k]] </pre>
1	<p>Two color tables and two opacity tables are used to look up the colors and alpha channel values of two different data sets:</p> <pre> red[i, j, k] = (RGB_TABLE0[DATA0[i, j, k], 0] * RGB_TABLE1[DATA1[i, j, k], 0]) / 255 green[i, j, k] = (RGB_TABLE0[DATA0[i, j, k], 1] * RGB_TABLE1[DATA1[i, j, k], 1]) / 255 blue[i, j, k] = (RGB_TABLE0[DATA0[i, j, k], 2] * RGB_TABLE1[DATA1[i, j, k], 2]) / 255 alpha[i, j, k] = (OPACITY_TABLE0[DATA0[i, j, k]] * OPACITY_TABLE1[DATA1[i, j, k]]) / 255 </pre>
2	<p>Four different data sets are used to provide the values to look up in the <i>RGB_TABLE0</i> and <i>OPACITY_TABLE0</i> properties to give the colors and alpha channel values:</p> <pre> red[i, j, k] = RGB_TABLE0[DATA0[i, j, k], 0] green[i, j, k] = RGB_TABLE0[DATA1[i, j, k], 1] blue[i, j, k] = RGB_TABLE0[DATA2[i, j, k], 2] alpha[i, j, k] = OPACITY_TABLE0[DATA3[i, j, k]] </pre>

Since a polygonal mesh was already visualized in Section 9.4, “A 3-dimensional example” [p. 249], a slight twist is given in this example: create a given number of isosurfaces of a volume and display those in a single visualization. For example data, use the thunderstorm data in the demo data of the IDL distribution:

```

IDL> restore, filename=filepath('storm25.sav', subdir=['examples', 'demo', 'demodata']), /verbose
% RESTORE: Portable (XDR) compressed SAVE/RESTORE file.
% RESTORE: Save file written by paulcs@C00TER, Wed Jun 28 17:03:46 2000.
% RESTORE: IDL version <Development build of Sun Jun 11 23:31:39 MDT 2000> (Win32, x86).
% RESTORE: Restored variable: P.
% RESTORE: Restored variable: T.
% RESTORE: Restored variable: U.
% RESTORE: Restored variable: V.

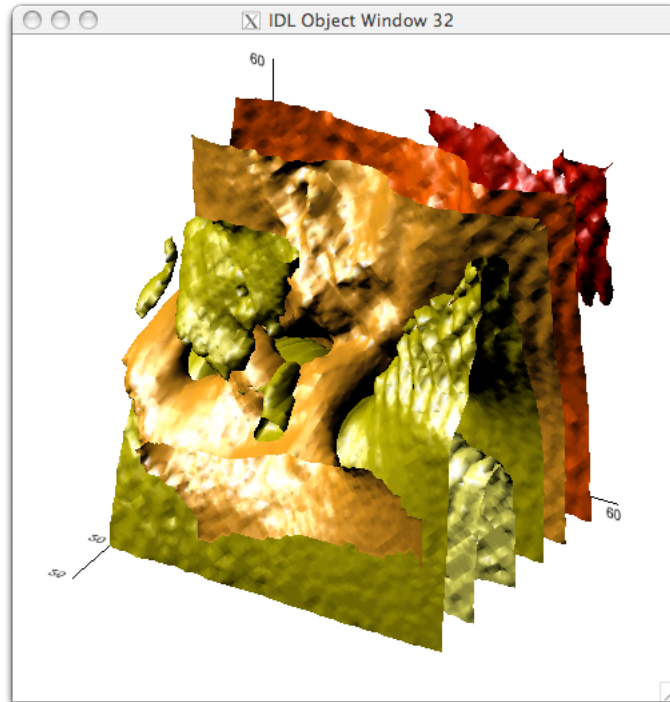
```

```
% RESTORE: Restored variable: W.
```

Let's display 9 isosurfaces of the p variable:

```
IDL> mg_ogisosurface_example, p, 9, /shading
```

The output should look like



Isosurfaces of the pressure data from storm25.sav save file is shown using the MG_OGISOSURFACE_EXAMPLE procedure. The “Std Gamma-II” color table is used to show the values of the isosurfaces.

The implementation is shown below:

```
pro mg_ogisosurface_example, volume, n, renderer=renderer, _extra=e
  compile_opt strictarr

  ; first and last values won't get drawn so add two to make up for it
  _n = n + 2

  dims = size(volume, /dimensions)
  xr = [0, dims[0]]
  yr = [0, dims[1]]
  zr = [0, dims[2]]

  xc = mg_linear_function(xr, [-0.6, 0.6])
  yc = mg_linear_function(yr, [-0.6, 0.6])
  zc = mg_linear_function(zr, [-0.6, 0.6])

  viewgroup = obj_new('IDLgrViewGroup')

  view = obj_new('IDLgrView')
  viewgroup->add, view

  model = obj_new('IDLgrModel')
```

```

view->add, model

palette = obj_new('IDLgrPalette')
viewgroup->add, palette
palette->loadCT, 5

maxV = max(volume, min=minV)
values = (maxV - minV) * findgen(_n) / (_n - 1L) + minV
colors = bytscl(values)

for i = 0L, _n - 1L do begin
    isosurface, volume, values[i], vertices, polygons
    if (polygons[0] lt 0L) then continue
    polygon = obj_new('IDLgrPolygon', vertices, polygons=polygons, $
        palette=palette, color=colors[i], $
        _extra=e)
    model->add, polygon
    polygon->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc
endfor

lightModel = obj_new('IDLgrModel')
view->add, lightModel

light = obj_new('IDLgrLight', type=2, location=[-1, 1, 1])
lightModel->add, light

model->rotate, [0, 1, 0], -30
model->rotate, [1, 0, 0], 30

xaxis = obj_new('IDLgrAxis', direction=0, range=xr, /exact, $
    location=[xr[0], yr[0], zr[0]], ticklen=0.015)
model->add, xaxis

yaxis = obj_new('IDLgrAxis', direction=1, range=yr, /exact, $
    location=[xr[0], yr[0], zr[0]], ticklen=0.015)
model->add, yaxis

zaxis = obj_new('IDLgrAxis', direction=2, range=zr, /exact, $
    location=[xr[0], yr[0], zr[0]], ticklen=0.015)
model->add, zaxis

xaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc
yaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc
zaxis->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc

window = obj_new('IDLgrWindow', dimensions=[500, 500], $
    graphics_tree=viewgroup, renderer=renderer)
window->draw
end

```

The `MESH_` routines in Table 6.15, “Routines to handle meshes” [p. 171] can be useful for dealing with polygonal meshes.

Map outlines can be represented by *IDLgrPolyline* objects and computed using the mapping projection routines. The `MG_OGMAP_EXAMPLE` procedure uses state outlines from a shapefile to display a map of the United States using any of the available map projections:

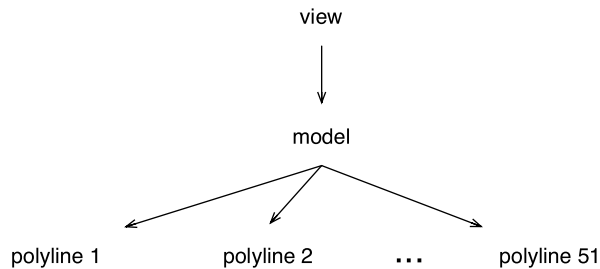
```
IDL> mg_ogmap_example, 'stereographic', color=[200, 100, 100]
```

This produces the following:



Stereographic projection of the US state outlines using the object graphics program MG_OGMAP_EXAMPLE.

The object graphics hierarchy for the example contains a view, model, and an *IDLgrPolyline* for each state and the District of Columbia:



The parameter to MG_OGMAP_EXAMPLE is a string listing the projection to use (the same as MAP_PROJ_INIT's argument). Keywords from *IDLgrPolyline::init* are accepted:

```

pro mg_ogmap_example, projection, renderer=renderer, _extra=e
  compile_opt strictarr

  _projection = n_elements(projection) eq 0L ? 'cylindrical' : projection

  map = map_proj_init(_projection)

  statesFilename = filepath('states.shp', subdir=['examples', 'data'])
  states = obj_new('IDLffShape', statesFilename)
  
```

```

view = obj_new('IDLgrView')
model = obj_new('IDLgrModel')
view->add, model

states->getProperty, n_entities=nEntities
for s = 0L, nEntities - 1L do begin
    state = states->getEntity(s)

    conn = [0]
    for p = 0, state.n_parts - 1L do begin
        startInd = (*state.parts)[p]
        endInd = p eq state.n_parts - 1 $
            ? state.n_vertices $
            : (*state.parts)[p + 1]

        conn = [conn, endInd - startInd, lindgen(endInd - startInd) + startInd]
    endfor

    xy = map_proj_forward(*state.vertices, map_structure=map)
    xyMax = max(xy, dimension=2, min=xyMin)
    if (n_elements(viewMax) gt 0) then begin
        viewMax >= xyMax
        viewMin <= xyMin
    endif else begin
        viewMax = xyMax
        viewMin = xyMin
    endelse

    statePoly = obj_new('IDLgrPolyline', xy, polylines=conn[1:*], $
                        _extra=e)
    model->add, statePoly

    states->destroyEntity, state
endfor

obj_destroy, states

sz = viewMax - viewMin
view->setProperty, viewplane_rect=[viewMin, sz]

window = obj_new('IDLgrWindow', graphics_tree=view, title=_projection, $
                 dimensions=500 * [1, sz[1] / sz[0]], renderer=renderer)
window->draw
end

```

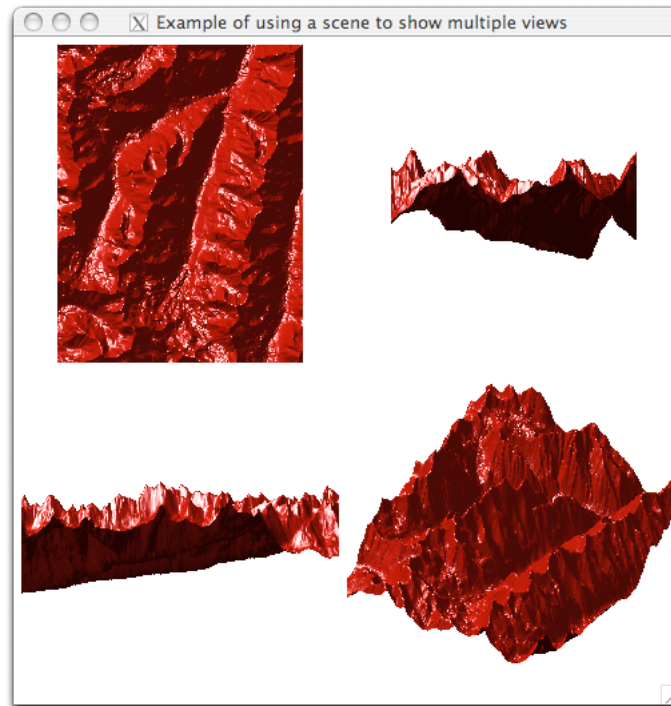
The final example in this section shows several views of a surface in a single scene. Using several views in one scene allows a different coordinate system for each view, making each one independent of the others. This is useful when combining different visualizations into a single display. The example program takes a 2-dimensional array and displays it as a surface with several different orientations. To see the output from the example, do

```

IDL> restore, filename=filepath('marbells.dat', subdir=['examples', 'data']), /verbose
% RESTORE: Portable (XDR) SAVE/RESTORE file.
% RESTORE: Save file written by ddirks@ENGDDIRKS, Fri Oct 05 12:52:26 2007.
% RESTORE: IDL version 7.0 <Dev build Thu Oct 4 23:07:16 MDT 2007> (Win32, x86).
% RESTORE: Restored variable: ELEV.
IDL> mg_ogscene_example, elev, findgen(350) * 10, findgen(450) * 10, /isotropic

```

This should display the following scene:



Display of example data file *marbells.dat* using *MG_OGSCENE_EXAMPLE*.

A 2-dimensional array z , as well as optional 1-dimensional arrays x and y to specify the axes values, are passed as parameters to the example routine. A helper routine is called four times to create the the four separate views of the surface. Each time the model inside the returned view is rotated to a different perspective.

```
pro mg_ogscene_example, z, x, y, isotropic=isotropic
  compile_opt strictarr

  _z = n_elements(z) eq 0L ? hanning(20, 20) : z
  dimensions = [0.5, 0.5]

  scene = obj_new('IDLgrScene')

  view1 = mg_ogscene_example_createview(_z, datax=x, datay=y, $
                                         model=model1, isotropic=isotropic, $
                                         location=[0.0, 0.5], $
                                         dimensions=dimensions)

  scene->add, view1

  view2 = mg_ogscene_example_createview(_z, datax=x, datay=y, $
                                         model=model2, isotropic=isotropic, $
                                         location=[0.5, 0.5], $
                                         dimensions=dimensions)

  scene->add, view2
  model2->rotate, [1, 0, 0], -90

  view3 = mg_ogscene_example_createview(_z, datax=x, datay=y, $
                                         model=model3, isotropic=isotropic, $
                                         location=[0.0, 0.0], $
                                         dimensions=dimensions)
```

```

scene->add, view3
model3->rotate, [1, 0, 0], -90
model3->rotate, [0, 1, 0], 90

view4 = mg_ogscene_example_createview(_z, datax=x, datay=y, $
                                     model=model4, isotropic=isotropic, $
                                     location=[0.5, 0.0], $
                                     dimensions=dimensions)

scene->add, view4
model4->rotate, [1, 0, 0], -90
model4->rotate, [0, 1, 0], -30
model4->rotate, [1, 0, 0], 45

win = obj_new('IDLgrWindow', dimensions=[500, 500], graphics_tree=scene, $
              title='Example of using a scene to show multiple views')
win->draw
end

```

The helper routine `MG_SCENE_EXAMPLE_CREATEVIEW` returns a view with a complete hierarchy of a model containing the surface as well as a model containing directional and ambient lights. The model is also returned via a keyword so that it can be rotated in the main routine. The *LOCATION* and *DIMENSIONS* keywords describe the view's location and size within the scene.

```

function mg_scene_example_createview, z, datax=datax, datay=datay, model=model, $
                                     location=location, dimensions=dimensions, $
                                     isotropic=isotropic

  compile_opt strictarr

  view = obj_new('IDLgrView', location=location, dimensions=dimensions, units=3)

  model = obj_new('IDLgrModel')
  view->add, model

  surf = obj_new('IDLgrSurface', z, datax=datax, datay=datay, style=2, $
                           color=[140, 14, 15], bottom=[60, 6, 6])
  model->add, surf

  lightmodel = obj_new('IDLgrModel')
  view->add, lightmodel

  dirLight = obj_new('IDLgrLight', type=2, location=[-1, 1, 1])
  lightmodel->add, dirLight

  ambLight = obj_new('IDLgrLight', type=0, intensity=0.4)
  lightmodel->add, ambLight

  m = 0.95
  surf->getProperty, xrange=xr, yrange=yr, zrange=zr
  maxRange = (xr[1] - xr[0]) > (yr[1] - yr[0]) > (zr[1] - zr[0])
  print, xr, yr, zr
  if (keyword_set(isotropic)) then begin
    xc = mg_linear_function([-0.5, 0.5] * maxRange + (xr[0] + xr[1]) / 2.0, [-m, m])
    yc = mg_linear_function([-0.5, 0.5] * maxRange + (yr[0] + yr[1]) / 2.0, [-m, m])
    zc = mg_linear_function([-0.5, 0.5] * maxRange + (zr[0] + zr[1]) / 2.0, [-m, m])
  endif else begin
    xc = mg_linear_function(xr, [-m, m])
    yc = mg_linear_function(yr, [-m, m])
    zc = mg_linear_function(zr, [-m, m])
  endif

```

```

endelse

surf->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc

return, view
end

```

The *ISOTROPIC* keyword specifies whether the data should be scaled to fit into each dimension or whether a single scaling should be used for all dimensions.

9.8. Texture mapping and transparency

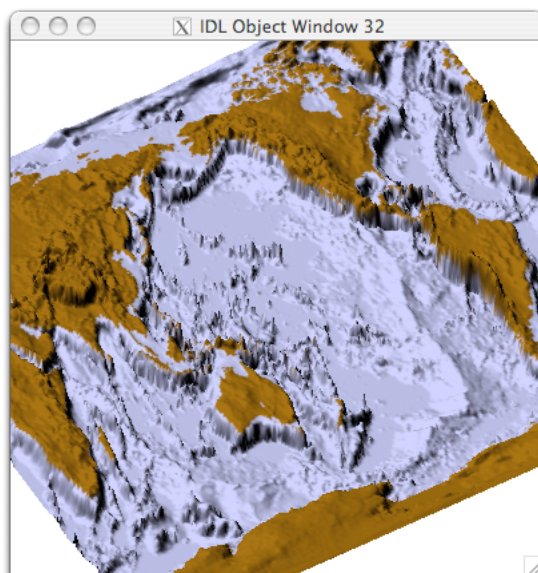
Surface and polygon graphics atoms can have images morphed onto them using a technique called texture mapping. Optionally, texture map images can contain an extra channel, called the *alpha channel*, which specifies the level of transparency for each pixel of the image. If this level is the same for all pixels, the *ALPHA_CHANNEL* property can instead be used to set the level with a single value.

Placing a texture map on a surface object is relatively easy. An *IDLgrImage* object is created as the texture map and passed as the value of the *TEXTURE_MAP* property of the *IDLgrSurface* object. Additionally, the *TEXTURE_INTERP* property can be set to specify nearest neighbor interpolation (0, the default) or bilinear interpolation (1) of the texture map to the surface.

Using data in the example data of the IDL distribution, the following example *MG_SURFACETEXTURE_DEMO* uses the world elevation data to create a surface and the continent mask to color code the surface as land or water:

```
IDL> mg_surfacetexture_demo
```

The result is displayed below.



The continent_mask.dat data set is used as a texture map over the worldelev.dat DEM in the MG_SURFACETEXTURE_DEMO example program.

Let's go through the lines of *MG_SURFACETEXTURE_DEMO*. First, the DEM and mask are read from binary data files:


```

pro mg_surfacetexture_demo, renderer=renderer
  compile_opt strictarr

  demFilename = file_which('worldelv.dat')
  dem = read_binary(demFilename, data_type=1, data_dims=[360, 360])

  maskFilename = file_which('continent_mask.dat')
  mask = read_binary(maskFilename, data_type=1, data_dims=[360, 360])

```

The object graphics hierarchy begins with a viewgroup, view, and model; the viewgroup is useful because the texture map image object will not be part of the object graphics hierarchy, so adding the texture map image to the viewgroup will clean it up automatically.

```

  viewgroup = obj_new('IDLgrViewGroup')

  view = obj_new('IDLgrView')
  viewgroup->add, view

  model = obj_new('IDLgrModel')
  view->add, model

```

The texture map image is a normal *IDLgrImage*. It is *landColor* over the land and *waterColor* over the water:

```

  landColor = [150, 100, 20]
  waterColor = [200, 200, 255]

  textureImage = bytarr(3, 360, 360)

  textureImage[0, *, *] = mask * landColor[0] + (1 - mask) * waterColor[0]
  textureImage[1, *, *] = mask * landColor[1] + (1 - mask) * waterColor[1]
  textureImage[2, *, *] = mask * landColor[2] + (1 - mask) * waterColor[2]

  texture = obj_new('IDLgrImage', textureImage)
  viewgroup->add, texture

```

Next, the surface graphics atom is created and added to the hierarchy. All the keywords used in this example are significant for working with texture maps. It is important to note that the color of the surface will be blended with the texture map, so in most cases the color should be set to white. Also, while texture maps can be mapped onto wire meshes and other style surfaces, smooth surfaces (*style=2*) are usually what is desired. Finally, the texture map is set via the *TEXTURE_MAP* keyword and bilinear interpolation is used:

```

  surface = obj_new('IDLgrSurface', dem, style=2, $
    color=[255, 255, 255], $
    texture_map=texture, texture_interp=1)
  model->add, surface

```

A directional light is added to its own model in the graphics hierarchy:

```

  lightmodel = obj_new('IDLgrModel')
  view->add, lightmodel

  light = obj_new('IDLgrLight', type=2, location=[-1, 1, 1])
  lightmodel->add, light

```

The surface is scaled to fit in the view volume using the *MG_LINEAR_FUNCTION* function introduced previously:

```

  surface->getProperty, xrange=xr, yrange=yr, zrange=zr
  xc = mg_linear_function(xr, [-1.2, 1.2])
  yc = mg_linear_function(yr, [-1.2, 1.2])
  zc = mg_linear_function(zr, [-0.1, 0.1])

```

```
surface->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc
```

The model is rotated to enhance the 3-dimensional aspect of the surface:

```
model->rotate, [1, 0, 0], -90
model->rotate, [0, 1, 0], 30
model->rotate, [1, 0, 0], 50
```

Finally, a window is created to display the scene. The viewgroup is set as the *GRAPHICS_TREE* of the window so that the graphics hierarchy will be destroyed when the window is deleted.

```
win = obj_new('IDLgrWindow', dimensions=[400, 400], graphics_tree=viewgroup, renderer=renderer)
win->draw
end
```

It is easier to texture map an image onto a surface than a polygon; a later example will map an image onto a non-rectangular polygon.

Graphics atoms can be made transparent in two different methods: using a texture map with an alpha channel (for surface and polygon atoms) or using the *ALPHA_CHANNEL* keyword (for nearly all atoms). In the above example, the *textureImage* could be specified as a four channel image with the fourth channel (the alpha channel) set to 0 for ocean and 255 for land:

```
textureImage[3, *, *] = mask * 255
```

This would produce a surface where only the land was visible. Alternatively, if the entire surface was to be made equally transparent, then adding *alpha_channel=0.2* to the *IDLgrSurface* creation would be sufficient.

One issue to keep in mind when using transparency is that graphics items are rendered in the order they are added to the object graphics tree. So a transparent item should be added *after* the items behind it. For an example of this problem, run the *MG_RENDER_ORDER* example:

```
IDL> mg_render_order
```

Both squares in this example are 50% transparent. The blue square is added first, so it appears through the red square when the red square is in the front. But rotating the squares to place the blue square in front causes the problem: the red square is not visible through the blue square. To show a solution to this problem, try

```
IDL> mg_render_order, /swap
```

With the *SWAP* keyword set, the order of the squares is changed by *MG_RENDER_ORDER* as they are rotated. See the code in *mg_render_order.pro* for more details of this example.

When using a texture map with an *IDLgrPolygon*, the same type of *IDLgrImage* texture map must be created as for *IDLgrSurface* objects. In addition, the *TEXTURE_COORD* properties should be set to texture coordinates that define the mapping from the image to the polygon. The texture coordinates are a 2 by *nverts* array, where *nverts* is the number of vertices in the polygon. For each vertex of the polygon, a location in the image is specified using normalized coordinates, i.e., the lower left hand corner of the image is (0.0, 0.0) and the upper right hand corner is (1.0, 1.0). Coordinates outside the range of 0–1 indicate repeated tilings. For example, the following texture coordinates would map a full image onto a rectangle:

```
texture_coord=[[0., 0.], [1., 0.], [1., 1.], [0., 1.]]
```

The following texture coordinates would map four copies of the image in a 2 by 2 grid onto the rectangle:

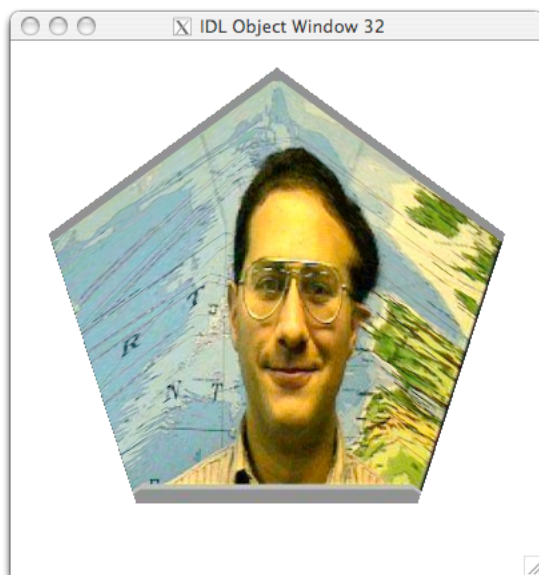
```
texture_coord=[[0., 0.], [2., 0.], [2., 2.], [0., 2.]]
```

It's as if the entire plane is tiled with the original image in the range 0–1 for both directions and the *TEXTURE_COORD* determines how much will be revealed.

The MG_TEXTUREMAP_DEMO procedure is an example of texture mapping an image onto a non-rectangular polygon, a pentagon.

```
IDL> mg_texturemap_demo
```

The result is shown below:



The example program MG_TEXTUREMAP_DEMO maps people.jpg in the example data of the IDL distribution onto a pentagon.

A viewgroup is again used as the root of the graphics tree so that the image object used as a texture map can be added to it to ensure it is cleaned up properly since it is not in the object graphics hierarchy. The beginning of the routine simply sets up the graphics hierarchy:

```
pro mg_texturemap_demo, renderer=renderer
  compile_opt strictarr

  ; viewgroup needed to properly cleanup texture map image
  viewgroup = obj_new('IDLgrViewGroup')

  view = obj_new('IDLgrView', color=[0, 0, 0])
  viewgroup->add, view

  model = obj_new('IDLgrModel')
  view->add, model
```

Next, create the texture map image as a normal image object, but add it to the viewgroup because it won't be part of the normal graphics tree:

```
f = filepath('people.jpg', subdir=['examples', 'data'])
ali = read_image(f)
texture = obj_new('IDLgrImage', ali)
viewgroup->add, texture
```

To define the pentagon, it is easiest to specify the location of the vertices in polar coordinates and convert them to rectangular coordinates:

```
r = fltarr(5) + 0.9
theta = (findgen(5) * 360 / 5 + 90.0) * !dtor
```

```
xy = cv_coord(from_polar=transpose([[theta], [r]]), /to_rect)
```

The texture coordinates will map four of the vertices of the pentagon to the four corners of the image. The fifth vertex, the one at the top, will be mapped to a point at the upper middle, (0.5, 1.0) in normal coordinates, of the image:

```
tcoords = [[0.5, 1.0], [0.0, 1.0], [0.0, 0.0], [1.0, 0.0], [1.0, 1.0]]
```

The texture map image and texture coordinates are passed to the creation of the *IDLgrPolygon* object:

```
polygon = obj_new('IDLgrPolygon', xy, polygons=[5, 0, 1, 2, 3, 4], $  
                 color=[255, 255, 255], $  
                 texture_map=texture, texture_interp=1, $  
                 texture_coord=tcoords)  
model->add, polygon
```

It is important to make sure the color of the polygon is set to white to prevent blending of the texture map with the underlying color of the polygon. Finally, create a graphics window and have it draw the graphics hierarchy:

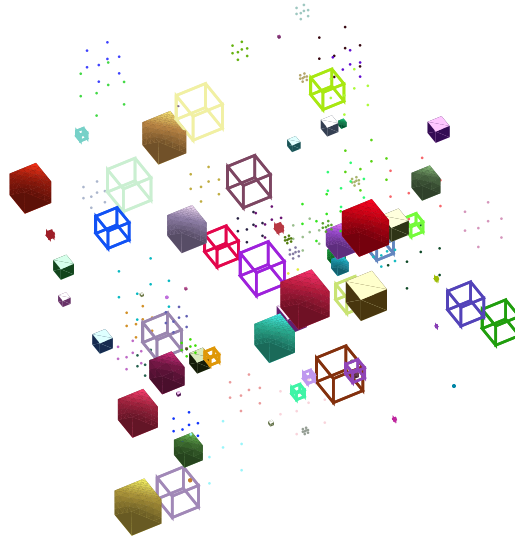
```
win = obj_new('IDLgrWindow', dimensions=[400, 400], graphics_tree=viewgroup, renderer=renderer)  
win->draw  
end
```

9.9. Inheriting from an *IDLgr* class: *MGgrCube*

The classes in the IDL object graphics library can be extended through inheritance like any other class. In this section, a *MGgrCube* class will be created that inherits from *IDLgrPolygon*. This cube could be used as a 3-dimensional plotting symbol or an indicator of spatial extent of a 3-dimensional object. To run the demo program, type the following:

```
IDL> .run mggrcube__define
```

This example should display 100 cubes with random locations, sizes, colors, and styles as below:



The `MGGRCUBE__DEFINE` procedure specifies that the *MGgrCube* class inherits from *IDLgrPolygon* and is described by *CENTER* and *SIDE* properties:

```
pro mggrcube__define
  compile_opt strictarr

  define = { MGgrCube, inherits IDLgrPolygon, $
             center: fltarr(3), $
             side: 0.0 $
           }
end
```

Properties from *IDLgrPolygon* will also be made available.

The *init* method calls the cube's parent's *init* method, *IDLgrPolygon::init*. It then saves away its own property values, using defaults if none are provided. Finally, it computes the actual vertices and polygons using the *recompute* method:

```
function mggrcube::init, center=center, side=side, _extra=e
  compile_opt strictarr

  if (~self->IDLgrPolygon::init(_extra=e)) then return, 0B

  ; save center and side, use defaults if necessary
  self.center = n_elements(center) eq 0 ? fltarr(3) : center
  self.side = n_elements(side) eq 0 ? 1.0 : side

  ; compute vertices and connectivity
  self->recompute
```

```

    return, 1B
end

```

The *_EXTRA* keyword allows properties of *IDLgrPolygon* to be set on the cube. Keyword inheritance will be used to allow access to all of *IDLgrPolygon*'s properties in the cube's *init*, *setProperty*, and *getProperty* methods.

The *cleanup* method is not strictly required here since it just calls its parent's *cleanup* method (which would happen automatically without a *MGgrCube::cleanup* method):

```

pro mggrcube::cleanup
    compile_opt strictarr

    self->IDLgrPolygon::cleanup
end

```

It is useful to put in the *cleanup* as a placeholder for future additions, but also to prevent IDL from searching for it.

The *setProperty* method sets the *CENTER* or *SIDE* properties and recomputes the vertices. Other properties of *IDLgrPolygon* are simply passed on to *IDLgrPolygon::setProperty* to be handled there.

```

pro mggrcube::setProperty, center=center, side=side, _extra=e
    compile_opt strictarr

    if (n_elements(center) gt 0) then begin
        self.center = center
        self->recompute
    endif

    if (n_elements(side) gt 0) then begin
        self.side = side
        self->recompute
    endif

    if (n_elements(e) gt 0) then begin
        self->IDLgrPolygon::setProperty, _strict_extra=e
    endif
end

```

The *_STRICT_EXTRA* used in the call to *IDLgrPolygon::setProperty* ensures that invalid property names will cause an error.

The *getProperty* method reports the values of the cube's properties, querying the parent *IDLgrPolygon* properties if necessary:

```

pro mggrcube::getProperty, center=center, side=side, _ref_extra=e
    compile_opt strictarr

    if (arg_present(center)) then begin
        center = self.center
    endif

    if (arg_present(side)) then begin
        side = self.side
    endif

    if (n_elements(e) gt 0) then begin
        self->IDLgrPolygon::getProperty, _strict_extra=e
    endif

```

end

Remember to use `_REF_EXTRA` in the routine header for output keywords such as in a `getProperty` method.

Finally, the cube vertices are specified in the `recompute` helper method:

```
pro mggrcube::recompute
  compile_opt strictarr

  ; create a list of -1's and 1's to represent vertices
  x = reform(rebin([-1, 1], 2, 4, /sample), 8)
  y = reform(rebin([-1, 1], 4, 2, /sample), 8)
  z = reform(rebin([-1, 1], 8, /sample), 8)

  ; scale them to center and side
  x = x * self.side / 2.0 + self.center[0]
  y = y * self.side / 2.0 + self.center[1]
  z = z * self.side / 2.0 + self.center[2]

  ; make the 3 by 8 array
  verts = transpose([x], [y], [z])

  ; make each face
  front = [4, 6, 7, 5, 4]
  back = [4, 3, 2, 0, 1]
  right = [4, 7, 3, 1, 5]
  left = [4, 2, 6, 4, 0]
  top = [4, 2, 3, 7, 6]
  bottom = [4, 4, 5, 1, 0]

  ; create the connectivity list
  polygons = [front, back, right, left, top, bottom]

  ; put the data into the polygon
  self->setProperty, data=verts, polygons=polygons
end
```

In a more efficient implementation, some of the temporary results above could be computed in the `init` method and reused as necessary.

An *MGgrCube* object can be placed into a standard graphics hierarchy, but it is simpler to use *XOBJVIEW* to view the cube for demonstration. Create a unit cube and display it:

```
IDL> cube = obj_new('MGgrCube', center=[0, 0], side=1.0)
IDL> xobjview, cube
```

The default black color of the cube makes it difficult to see its edges. While *XOBJVIEW* is still running, change the color with the `setProperty` method:

```
IDL> cube->setProperty, color=[255, 200, 0]
```

Refresh the display in *XOBJVIEW* by rotating the cube a bit or selecting *View > Refresh Display* from the menus. Destroy the cube when finished:

```
IDL> obj_destroy, cube
```

XOBJVIEW can be closed before or after the cube is freed.

9.10. Composite graphics classes

Composite graphics classes combine several atoms into a single entity. This new class can be treated as a normal graphics atom with a more specialized purpose. This is analogous to a compound widget, but the object-oriented nature of the object graphics system allows for transparent integration of the new class (compound widgets, for example, can be awkward to control after creation because `WIDGET_CONTROL` does not know about them).

The advantage of combining the atoms into a separate class is that higher-level properties can be set on the class without worrying about the details of how they are implemented by the properties of the individual atoms that comprise the composite graphics class. For example, the properties of items in a *IDLgrLegend* can be set without knowledge of the properties of the underlying polyline or text objects.

Two composite graphics classes are provided in the IDL library *IDLgrLegend* and *IDLgrColorbar*. The IDL source code for both of them is in the IDL library and can be examined.

The example presented in this section will be a lighting system composed of several lights: an ambient light and two directional lights from different locations. To see output from using *MGgrLightModel*, run the main-level example program included with the code:

```
IDL> .run mggrlightmodel__define
```

This creates a simple object graphics hierarchy containing a surface and uses an *MGgrLightModel* to light it.

The composite graphics class itself inherits from *IDLgrModel* and adds its components as children of itself:

```
pro mggrlightmodel__define
  compile_opt strictarr

  define = { MGgrLightModel, inherits IDLgrModel }
end
```

Currently, there are no properties to the *MGgrLightModel*, so there are empty *getProperty* and *setProperty* methods. A more robust implementation would allow the properties of the three lights to be controlled independently.

The cleanup responsibilities are passed pass along to the model (it will destroy its children, the lights). This method is not strictly needed until other cleanup tasks are required, but creating it now means it can be added to quickly.

```
pro mggrlightmodel::cleanup
  compile_opt strictarr

  self->idlgrmodel::cleanup
end
```

The *init* method creates the three lights and adds them to the object:

```
function mggrlightmodel::init, _extra=e
  compile_opt strictarr

  if (~self->IDLgrModel::init(_strict_extra=e)) then return, 0B

  ambient = obj_new('IDLgrLight', type=0, name='ambient', intensity=0.2)
  self->add, ambient

  directionall1 = obj_new('IDLgrLight', type=2, location=[-1.0, 1.0, 1.0], $
```



```

        name='directional1', intensity=0.5)
self->add, directional1

directional2 = obj_new('IDLgrLight', type=2, location=[0.0, 0.0, 1.0], $
        name='directional2', intensity=0.3)
self->add, directional2

return, 1B
end

```

The *directional2* light shines directly from the viewer to the origin, while the *directional1* provides some 3-dimensional perspective to the scene by being offset.

9.11. Sending output to other destinations

A object graphics scene can be sent to a graphics window, the system clipboard, a printer, a memory buffer, or a VRML file via the destinations listed in Table 9.5, “Object graphics destination classes” [p. 247]. Object graphics is device-independent, meaning that the object graphics hierarchy does *not* need to be changed depending on the output destination.

Two common formats to capture graphics output are raster image formats (like PNG, JPEG, etc.) or vector graphics formats (like Encapsulated PostScript). Vector graphics uses geometric objects like points and lines to describe graphics. This allows for smooth looking output even when zoomed into or shown in high-resolution.

Use the *IDLgrClipboard* as the destination class to get Encapsulated PostScript (EPS) output. The *draw* method for *IDLgrClipboard* has keywords that must be configured to produce EPS output. Set the *VECTOR* keyword, set the *POSTSCRIPT* keyword, and specify an output filename with the *FILENAME* keyword. For example, the following code snippet would send the object graphics hierarchy rooted at *scene* to the EPS file *scene.eps*:

```

clipboard = obj_new('IDLgrClipboard')
clipboard->draw, scene, /vector, /postscript, filename='scene.eps'
obj_destroy, clipboard

```

There are some keywords to further configure vector output; see the *VECT_SHADING*, *VECT_SORTING*, and *VECT_TEXT_RENDER_METHOD* keywords to *IDLgrClipboard::draw* in the online help.

The *IDLgrPDF* destination, added in IDL 8.0, is similar to the clipboard destination, but allows multiple pages of output to be created. A typical rendering to PDF would be like:

```

pdf = obj_new('IDLgrPDF')
pdf->addPage
pdf->draw, scene1
pdf->addPage, /landscape
pdf->draw, scene2
pdf->save, 'scenes.pdf'
obj_destroy, pdf

```

The size and orientation for each page can be set with the *addPage*, with the default being an 8.5 by 11 inch page in portrait orientation. Beyond the normal graphics destination properties, there are also several properties to control meta-data contained with the PDF file.

Creating a raster image of a scene can be done with either the *IDLgrWindow* or *IDLgrBuffer* destinations; using a window will display the graphics on the screen as well. There are two ways to get the output of a graphics scene as an image depending on what is to be done with the image: the *IMAGE_DATA* property or the *read* method. Use the

IMAGE_DATA property of either *IDLgrWindow* or *IDLgrBuffer* to get a standard IDL array that can be used as output to *WRITE_IMAGE* or its equivalents. For example, the following code snippet creates a 400 by 400 pixel PNG image file of the graphics hierarchy rooted at *scene*:

```
win = obj_new('IDLgrWindow', dimensions=[400, 400])
win->draw, scene
win->getProperty, image_data=im
write_png, 'output.png', im
obj_destroy, win
```

The *read* method of *IDLgrWindow* or *IDLgrBuffer* retrieves the current display as an *IDLgrImage*:

```
image = win->read()
```

In this case, *image* could then be queried for its properties, its data retrieved, or used in another graphics hierarchy.

9.12. Creating a new destination: *MGgrWindow3D*

It is possible to create new destination classes that render graphics hierarchies in new ways or to new formats. For example, new destination classes could write Scalable Vector Graphics (SVG), POV-Ray input files, or any of the various image file formats.

The example in this section, *MGgrWindow3d*, will display output in a standard graphics window, but will render it as an anaglyph, i.e., two superimposed images, one red and one cyan, that when viewed with red-cyan glasses produce a stereo effect. To see a simple result, try

```
IDL> .run mggrwindow3d__define
```

This will create a simple scene containing a surface and render it with *MGgrWindow3d* instead of *IDLgrWindow*. Red-cyan glasses are required to perceive the 3-dimensional effect of the output image.

The *MGgrWindow3d* will use a helper class, *MGgr3dConverter* to do the computations necessary to make the anaglyph: it rotates the top-level models in a scene, creates two images from these rotations, and combines the two images into a single red-cyan anaglyph. These calculations could be done multiple ways to produce other types of output: there are stereograms, autostereograms, or even output that requires special hardware for display. Therefore, this class encapsulates one way of doing the calculation, but other classes could be written to do the calculations other ways.

The converter class contains a separate object graphics hierarchy to pass along to the *MGgrWindow3d*; the *view* and *image* fields are part of this hierarchy. The *buffer* field is a destination that is used to render the results that are placed in the *image* field. The *eyeSeparation* field is the angle of separation in degrees between the “eyes” of the renderer.

```
pro mggr3dconverter__define
  compile_opt strictarr

  define = { MGgr3dConverter, $
    eyeSeparation: 0.0, $
    buffer: obj_new(), $
    view: obj_new(), $
    image: obj_new() $
  }
end
```

The object graphics hierarchy passed along to the *IDLgrWindow3d* as well as the buffer used to create the two images that are combined into the anaglyph are created in the *init* method.

```
function mggr3dconverter::init, eye_separation=eyeSeparation, $
```

```

                                dimensions=dimensions, picture=picture, _extra=e
compile_opt strictarr

self.eyeSeparation = n_elements(eyeSeparation) eq 0 ? 4.0 : eyeSeparation

self.buffer = obj_new('IDLgrBuffer', dimensions=dimensions, _extra=e)

self.view = obj_new('IDLgrView', viewplane_rect=[0, 0, dimensions])

model = obj_new('IDLgrModel')
self.view->add, model

self.image = obj_new('IDLgrImage')
model->add, self.image

return, 1
end

```

The graphics hierarchy and the buffer need to be cleaned up when done:

```

pro mggr3dconverter::cleanup
  compile_opt strictarr

  obj_destroy, [self.view, self.buffer]
end

```

There are two properties for the converter class: *EYE_SEPARATION* and *DIMENSIONS*. The *DIMENSIONS* property is actually stored in the buffer:

```

pro mggr3dconverter::getProperty, eye_separation=eyeSeparation, $
                                dimensions=dimensions
  compile_opt strictarr

  if (arg_present(eyeSeparation)) then begin
    eyeSeparation = self.eyeSeparation
  endif

  if (arg_present(dimensions)) then begin
    self.buffer->getProperty, dimensions=dimensions
  endif
end

```

The *DIMENSIONS* is passed along to the buffer as well as used to set the view's *VIEWPLANE_RECT* property:

```

pro mggr3dconverter::setProperty, eye_separation=eyeSeparation, $
                                dimensions=dimensions
  compile_opt strictarr

  if (n_elements(eye_separation) gt 0) then begin
    self.eyeSeparation = eyeSeparation
  endif

  if (n_elements(dimensions) gt 0) then begin
    self.view->setProperty, viewplane_rect=[0, 0, dimensions]
    self.buffer->setProperty, dimensions=dimensions
  endif
end

```

The *_combineImages* method combines left and right images into a single image where the left image is used for the red channel and the right image is used for the green and blue channels:

```

function mggr3dconverter::_combineImages, leftImage, rightImage
  compile_opt strictarr

  ; define combined_image to the correct size
  combinedImage = leftImage * 0B
  dims = size(leftImage, /dimensions)

  _leftImage = byte(total(fix(leftImage), 1) / 3)
  _rightRight = byte(total(fix(rightImage), 1) / 3)

  combinedImage[0, 0, 0] = Reform(_leftImage, 1, dims[1], dims[2])
  combinedImage[1, 0, 0] = Reform(_rightRight, 1, dims[1], dims[2])
  combinedImage[2, 0, 0] = Reform(_rightRight, 1, dims[1], dims[2])

  return, combinedImage
end

```

Note that we prefix the method name with an underscore to indicate that it is a private method, not intended to be called from outside of the other methods of the class. The `_rotateModels` method is a helper method that rotates top-level models the given number of degrees:

```

pro mggr3dconverter::_rotateModels, picture, degrees
  compile_opt strictarr

  ; if picture is a model then rotate it, but don't rotate models inside it
  if (obj_isa(picture, 'IDLgrModel')) then begin
    picture->rotate, [0, 1, 0], degrees
    return
  endif

  if (obj_isa(picture, 'IDL_Container')) then begin
    items = picture->get(/all, count=count)
    for i = 0L, count - 1 do begin
      self->_rotateModels, items[i], degrees
    endfor
  endif
end

```

The `convert` method is the only method besides the lifecycle and property methods that should be called from outside the class. It converts an arbitrary graphics hierarchy containing 3-dimensional objects to another object graphics hierarchy containing an anaglyph image:

```

function mggr3dconverter::convert, picture
  compile_opt strictarr

  ; rotate "top-level" models for left eye
  self->_rotateModels, picture, self.eyeSeparation / 2.

  ; draw picture to left eye buffer
  self.buffer->draw, picture

  ; get data out of left eye buffer
  oleftImage = self.buffer->read()
  oleftImage->getProperty, data=leftImage
  obj_destroy, oleftImage

  ; rotate "top-level" models for right eye
  self->_rotateModels, picture, - self.eyeSeparation

```

```

; draw picture to right eye buffer
self.buffer->draw, picture

; get data out of left eye buffer
orightImage = self.buffer->read()
orightImage->getProperty, data=rightImage
obj_destroy, orightImage

; rotate "top-level" models back to center
self->_rotateModels, picture, self.eyeSeparation / 2.

combinedImage = self->_combineImages(leftImage, rightImage)

self.image->setProperty, data=combinedImage

return, self.view
end

```

The outline of the *convert* method is simple: rotate to the right, take a snapshot, rotate back to the left, take a snapshot, combine the two snapshots into an anaglyph, reset the rotation, and return the anaglyph.

The actual destination *MGgrWindow3d* is a subclass of *IDLgrWindow*. It contains an *MGgr3dConverter* object to construct the anaglyph:

```

pro mggrwindow3d__define
  compile_opt strictarr

  define = { MGgrWindow3d, inherits IDLgrWindow, $
              converter: obj_new() $
            }
end

```

The *DIMENSIONS* property takes on a platform-dependent preference value for the default size of a graphics window if not set explicitly. A converter object is created at the same time as the window:

```

function mggrwindow3d::init, eye_separation=eyeSeparation, $
                                dimensions=dimensions, _extra=e
  compile_opt strictarr

  if (~self->IDLgrWindow::init(dimensions=dimensions, _extra=e)) then return, 0

  if (n_elements(dimensions) eq 0) then begin
    case strlowcase(!version.os_family) of
      'unix' : begin
        dims = [pref_get('idl_gr_x_width'), pref_get('idl_gr_x_height')]
      end
      'windows' : begin
        dims = [pref_get('idl_gr_win_width'), pref_get('idl_gr_win_height')]
      end
    endcase
    endif else dims = dimensions

  self.converter = obj_new('MGgr3dConverter', $
                            eye_separation=eyeSeparation, $
                            dimensions=dims, _extra=e)

  return, 1

```

```
end
```

The parent class *IDLgrWindow* must be cleaned up, along with the converter object:

```
pro mggrwindow3d::cleanup
  compile_opt strictarr

  self->idlgrwindow::cleanup
  obj_destroy, self.converter
end
```

The *draw* method sends the object graphics hierarchy to the converter object to receive back a new hierarchy containing the anaglyph image which is then drawn by the parent class' *draw* method.

```
pro mggrwindow3d::draw, picture
  compile_opt strictarr
  on_error, 2

  self->getProperty, graphics_tree=graphicsTree
  _picture = obj_valid(picture) ? picture : graphicsTree

  view = self.converter->convert(_picture)

  self->idlgrwindow::draw, view
end
```

The *EYE_SEPARATION* property is stored in the converter class, the rest of the properties are retrieved from the parent class:

```
pro mggrwindow3d::getProperty, eye_separation=eyeSeparation, _ref_extra=e
  compile_opt strictarr

  if (arg_present(eyeSeparation)) then begin
    self.converter->getProperty, eye_separation=eyeSeparation
  endif

  if (n_elements(e) gt 0) then begin
    self->IDLgrWindow::getProperty, _strict_extra=e
  endif
end
```

The *DIMENSIONS* property is intercepted so that it can be sent to both the parent *IDLgrWindow* and to the converter:

```
pro mggrwindow3d::setProperty, dimensions=dimensions, $
  eye_separation=eyeSeparation, _extra=e
  compile_opt strictarr

  self->IDLgrWindow::setProperty, _extra=e

  if (n_elements(dimensions) gt 0) then begin
    self->IDLgrWindow::setProperty, dimensions=dimensions
    self.converter->setProperty, dimensions=dimensions
  endif

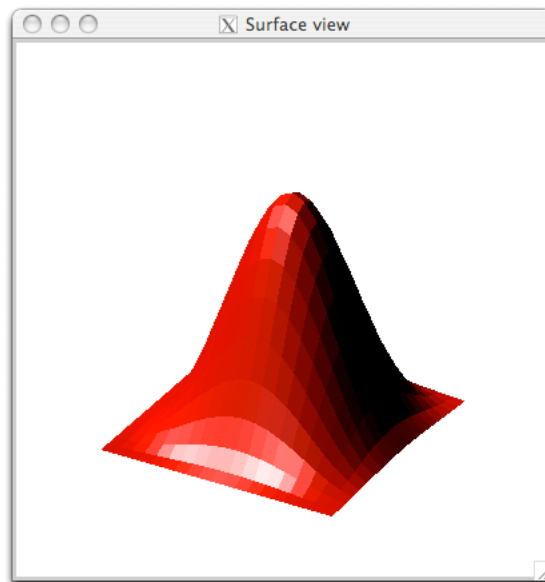
  if (n_elements(eye_separation) gt 0) then begin
    self.converter->setProperty, eye_separation=eyeSeparation
  endif
end
```

The converter handles the *EYE_SEPARATION* property.

9.13. Widgets and objects graphics: interaction

Object graphics combines well with widget programs. The more lengthy setup required for object graphics is not a major drawback in a widget program that already requires a fair amount of setup. Also, the persistent nature of the objects in the object graphics scene creates many possibilities for useful interactivity.

MG_SURFVIEW is a minimal program to display an object graphics scene in a widget program. It does not provide much functionality except to display a surface and allow it to be rotated interactively with the mouse. To rotate, hold the left mouse button down while moving the mouse. More features are added through the exercises for this section.



The MG_SURFVIEW program displaying the default data set.

There are only three routines for this program: MG_SURFVIEW to create the object graphic and widget hierarchies, MG_SURFVIEW_CLEANUP to clean up resources, and MG_SURFVIEW_EVENT to handle all the events generated in the program. All the routines are in a single file *mg_surfview.pro* and are discussed in the order that they would likely be written, not in the order that they should appear in the file. Let's step through the sections of the main routine, MG_SURFVIEW, starting with the routine header that indicates the routine accepts a single parameter, a 2-dimensional data set to display as a surface. The HANNING function is used to create an example data set if none is specified. The default graphics window size will be set to 400 by 400.

```
pro mg_surfview, z, renderer=renderer
  compile_opt strictarr

  ; default data for convenience during development
  _z = n_elements(z) eq 0 ? hanning(20, 20) : z

  xsize = 400
  ysize = 400
```

Next, the widget hierarchy is created and realized. The draw widget's *GRAPHICS_LEVEL* keyword is set to 2 to indicate that this draw widget will display object graphics instead of direct graphics (and this cannot be changed while the

program is running). It is possible to use subclasses of *IDLgrWindow* or *IDLitWindow* in a widget program. Along with setting *GRAPHICS_LEVEL* to 2, set the *CLASSNAME* keyword of the *WIDGET_DRAW* to the name of the class to use for the draw widget. Motion and button events need to be turned on to let the user rotate the surface.

```
; create widget hierarchy
tlb = widget_base(title='Surface view', /column, $
                  /tlb_size_events, uname='tlb')
draw = widget_draw(tlb, xsize=xsize, ysize=ysize, $
                  graphics_level=2, renderer=renderer, $ ; 2 => object graphics
                  /motion_events, /button_events, uname='draw')

widget_control, tlb, /realize
```

In a widget program that uses direct graphics in a draw widget, the following line would return a widget identifier in *owindow*. Because the *GRAPHICS_LEVEL* keyword is set to 2 in the creation of the *WIDGET_DRAW* above, an *IDLgrWindow* object reference is returned instead:

```
widget_control, draw, get_value=owindow
```

Next, a fairly simple object graphics hierarchy containing a surface is created. A directional light is created and placed in a separate model so that it does not move as the surface is rotated.

```
oview = obj_new('IDLgrView', color=[0, 0, 0])

omodel = obj_new('IDLgrModel')
oview->add, omodel

; style = 2 is filled
osurface = obj_new('IDLgrSurface', _z, style=2, $
                  color=[255, 0, 0], bottom=[100, 0, 0])
omodel->add, osurface

olightmodel = obj_new('IDLgrModel')
oview->add, olightmodel

; type = 2 is a directional light; shines from [-1, 1, 1] to [0, 0, 0]
olight = obj_new('IDLgrLight', type=2, location=[-1, 1, 1])
olightmodel->add, olight
```

Again, we will use the coordinate conversion functions to scale the surface into the view volume. Also, the model is rotated to better show the structure of the surface:

```
osurface->getProperty, xrange=xr, yrange=yr, zrange=zr
xc = norm_coord(xr)
xc[0] -= 0.5
yc = norm_coord(yr)
yc[0] -= 0.5
zc = norm_coord(zr)
zc[0] -= 0.5
osurface->setProperty, xcoord_conv=xc, ycoord_conv=yc, zcoord_conv=zc

; set a nice original orientation
omodel->rotate, [1, 0, 0], -90
omodel->rotate, [0, 1, 0], -30
omodel->rotate, [1, 0, 0], 30

owindow->draw, oview
```

The hierarchy is now completed and the scene is drawn to the window. The *Trackball* object does the calculations to rotate the surface. It computes transformation matrices representing rotations from the mouse motions of the user. It is

instantiated with a *center* to rotate about and *radius* determining the distance the mouse must move to make a single rotation of the surface. Here, the center of rotation is chosen to be the center of the draw widget and the radius is half the maximum of the horizontal or vertical sizes of the draw widget.

```
otrack = obj_new('Trackball', [xsize, ysize] / 2, (xsize > ysize) / 2)
```

The *state* structure contains a few object references: the view and window are necessary to redraw the graphics scene while the model and trackball are needed to rotate the surface. The draw widget identifier is needed for resizing. A pointer to this structure is created and stored in the top-level base's UVALUE in the standard technique for widget program data.

```
; setup state structure
state = { oview: oview, $
          omodel: omodel, $
          owindow: owindow, $
          otrack: otrack, $
          draw: draw $
        }
pstate = ptr_new(state, /no_copy)
widget_control, tlb, set_uvalue=pstate
```

The last thing to do in the main routine is start the program using XMANAGER:

```
; start XMANAGER
xmanager, 'mg_surfview', tlb, /no_block, $
          event_handler='mg_surfview_event', $
          cleanup='mg_surfview_cleanup'
end
```

Because the *NO_BLOCK* keyword is set, it is important to put cleanup code in a routine specified with the *CLEANUP* keyword; code directly after the XMANAGER call would be executed immediately (and the program would be cleaned up before it got going if the cleanup code were there).

The cleanup routine for this program, *MG_SURFVIEW_CLEANUP*, was registered with XMANAGER. It will be called with the top-level base widget identifier as its only argument when the program is terminated. Its job is to free any resources held by the application.

```
pro mg_surfview_cleanup, tlb
  compile_opt strictarr

  widget_control, tlb, get_uvalue=pstate

  obj_destroy, [(pstate).otrack, (pstate).oview]
  ptr_free, pstate
end
```

The *Trackball* object is not part of the object graphics hierarchy, so it must be freed separately from the *IDLgrView* object which destroys all the objects in the hierarchy.

The event handler does the work required when the user interacts with the program. For *MG_SURFVIEW*, there are only two possible user actions currently: rotating the surface and resizing the widgets. The resize case is fairly typical of widget resizing code except that the *Trackball* object must be told the new size (with the *Trackball::reset* method) so that it can continue to perform its calculations accurately. The resize code falls under the “tlb” case.

```
pro mg_surfview_event, event
  compile_opt strictarr

  widget_control, event.top, get_uvalue=pstate
```

```

uname = widget_info(event.id, /uname)

case uname of
  'tlb' : begin
    ; collect geometry information
    tlbG = widget_info(event.top, /geometry)

    ; calculate new draw widget size
    new_x = event.x - 2 * tlbG.xpad
    new_y = event.y - 2 * tlbG.ypad

    ; set draw widget size
    widget_control, (*pstate).draw, xsize=new_x, ysize=new_y

    ; refresh graphics
    (*pstate).owindow->draw, (*pstate).oview

    ; reset trackball for the new size draw widget
    (*pstate).otrack->reset, [new_x, new_y] / 2, (new_x > new_y) / 2
  end
  'draw' : begin
    ; motion and button events must be sent to trackball; it will set update
    ; if the model needs to be rotated
    update = (*pstate).otrack->update(event, transform=rotation)
    if update then begin
      (*pstate).omodel->getProperty, transform=transform
      (*pstate).omodel->setProperty, transform=transform # rotation
      (*pstate).owindow->draw, (*pstate).oview
    endif
  end
endcase
end

```

The code handling the rotating of the surface is a standard way of handling a *Trackball* object. The *Trackball::update* method returns whether the user is trying to rotate the surface, i.e., holding the left mouse button while moving the mouse. If so, the *TRANSFORM* keyword will return a transformation matrix that represents the rotation. The code inside the if statement gets the current transformation matrix, performs the rotation by multiplying by the rotation matrix, puts the resulting transformation matrix back, and redraws the screen. It is common to forget to redraw the window, but the changes will not be seen if it is not redrawn.

Cases can easily be added to the event handler as new widgets are added to the widget hierarchy. As more cases are added and the event handler routine becomes longer, it would be beneficial to move the code for the cases to individual routines that are simply called in the case statement. In larger applications, the event handler is usually just a dispatcher to many other routines that actually do the work.

Displaying an object graphics scene in a widget program opens up the possibility to quickly add many powerful features. The fact that the properties of the objects in the hierarchy are persistent, can be queried, and can be selectively changed creates this power. Currently, MG_SURFVIEW is a framework to which many more interesting features can be added.

Exercises

The following exercises all describe modifications to MG_SURFVIEW demonstrating how easy it is to add powerful interactive features to an object graphics scene in a widget program. Many of these exercises can be done in a few lines of code, others are more involved.

1. Display the data coordinates under the mouse in a status bar bottom the draw widget. Use the *IDLgrWindow::pickData* method to find the coordinates. (The *IDLgrWindow::select* method can be used to find the the graphics atoms under the cursor in a more complicated graphics scene, but that is not necessary in this program since there is only one graphics atom.)
2. Make a double-click in the graphics window reset the surface to its original orientation.
3. Add a toolbar above the draw widget with a button to output the contents of the draw widget to a PNG file. See Section 9.11, “Sending output to other destinations” [p. 283] about getting an image of a graphics scene. Make sure to not leak any memory, i.e., cleanup any newly created objects.
4. Rotate the light instead of the surface when the user holds the shift key down while doing the normal rotation with the mouse.
5. Use the keyboard arrow keys to rotate the surface (left and right keys) and to zoom in/out (up and down keys). Set the *KEYBOARD_EVENTS* keyword of *WIDGET_DRAW* to 2 to generate keyboard events. Modify the event handler to handle these new events.
6. Rotating graphics interactively can be slow if there is a lot of data. One way to improve the response is to simplify the graphics so that they can be rendered faster. Change the style of the surface to mesh while the user is rotating it.

9.14. Widgets and object graphics: tiled imagery

The *MG_TILEJP2* example widget program uses the tiling capabilities of *IDLgrImage* and *IDLgrWindow* added in IDL 6.2 to view large images without reading all the data of the image. There are no features in this application except navigation around a large image.

The controls for navigating the image are simple: drag the image with the mouse (or use the arrow keys), zoom in/out with the mouse scroll wheel (or page up/down), or reset the view with the home key. The application is shown below.



Using `MG_TILEJP2` to display a JPEG2000 version of the `ohare.jpg` file in the example data of the IDL distribution.

This program uses JPEG2000 files because it is easy to pull out a section from the file. It would be possible to modify this program to use TIFF, MrSID, or even band-sequential binary files.

The main routine `MG_TILEJP2` is responsible for all initialization; in this case, creating the widget tree, the object graphics hierarchy, and the *state* structure to store the data needed in the event handlers. The routine takes the JPEG2000 filename as an argument and checks it for validity:

```
pro mg_tilejp2, jp2filename, renderer=renderer
  compile_opt strictarr
  on_error, 2

  _jp2filename = (n_elements(jp2filename) eq 0) ? 'ohare.jp2' : jp2filename

  if (~file_test(_jp2filename)) then begin
    message, 'JPEG2000 file not found'
  endif
```

Instead of creating a single `IDLffJPEG2000` object and saving it away, `IDLffJPEG2000` objects will be created as needed and freed immediately. The dimensions of the JPEG2000 image and each tile in the image will be needed when creating the `IDLgrImage`, so an `IDLffJPEG2000` object is created to query for them:

```
objp2 = obj_new('IDLffJPEG2000', _jp2filename)
objp2->getProperty, dimensions=imageDims, tile_dimensions=jp2TileDims
obj_destroy, objp2
```

The widget hierarchy is extremely simple, just a top-level base and a draw widget to display the imagery. Remember, it is necessary to set `GRAPHICS_LEVEL` of `WIDGET_DRAW` to 2 to display object graphics. To setup the keyboard events correctly, set `KEYBOARD_EVENTS` to 2 (setting it to 1 just turns on alphabetic keyboard events). Also, the draw widget

is given input focus (with the *INPUT_FOCUS* of *WIDGET_CONTROL*) so that keyboard events are generated without the user having to first click on the draw widget.

```
windowDims = [500, 500]

; create widget hierarchy
tlb = widget_base(title='JPEG2000 tile viewer', /column, /tlb_size_events, $
    uname='tlb', xpad=0, ypad=0)
draw = widget_draw(tlb, xsize=windowDims[0], ysize=windowDims[1], $
    uname='draw', graphics_level=2, renderer=renderer, $
    /button_events, /motion_events, /wheel_events, $
    keyboard_events=2)

widget_control, tlb, /realize
widget_control, draw, /input_focus
widget_control, draw, get_value=owindow
```

The object graphics hierarchy is also simple—only view, model, and image objects are needed. Set the view's *VIEWPLANE_RECT* property to the size of the draw window so that one unit in the object graphics scene will correspond to one pixel in the draw widget:

```
; create object graphics hierarchy
oview = obj_new('IDLgrView', name='view', color=[0, 0, 0], $
    viewplane_rect=[0, 0, windowDims[0], windowDims[1]])

omodel = obj_new('IDLgrModel', name='model')
oview->add, omodel
```

The image requires several tiling specific keywords. Tiling is turned on with the *TILING* keyword; the image and tile dimensions values are passed along from the *IDLffJPEG2000* properties. The *TILE_LEVEL_MODE* keyword is set 1, indicating that the data source has an image pyramid, i.e., data of various resolutions depending on the zoom level. The *TILE_SHOW_BOUNDARIES* keyword can be useful when debugging, but is turned off now.

```
oimage = obj_new('IDLgrImage', name='image', $
    order=1, $
    /tiling, $
    tile_show_boundaries=0, $
    tile_level_mode=1, $ ; automatic mode for image pyramid
    tiled_image_dimensions=imageDims, $
    tile_dimensions=jp2TileDims)

omodel->add, oimage
```

A pointer is created to a *state* structure and stored in the top-level base's “uvalue” as usual. The window and view are required for redrawing, the window dimensions are needed for resizing, the JPEG2000 filename is needed for getting data, and the rest of the variables indicate the state of the panning/zooming.

```
; setup data for event handlers
state = { owindow: owindow, $
    oview: oview, $
    windowDims: windowDims, $
    buttonsDown: 0B, $
    curLoc: [0L, 0L], $
    pressLoc: [0L, 0L], $
    zoomLevel: 0L, $
    zoomFactor: 1.0, $
    jp2filename: _jp2filename $
}
pstate = ptr_new(state, /no_copy)
```

```
widget_control, tlb, set_uvalue=pstate
```

Before starting, the current location of the image is drawn (the lower left corner of the image since *curLoc* starts at [0, 0]):

```
mg_tilejp2_refresh, pstate
```

As usual, XMANAGER starts the event loop, declares the routine to handle events, and specifies the routine to handle the cleanup for the program:

```
xmanager, 'mg_tilejp2', tlb, /no_block, $
    event_handler='mg_tilejp2_event', $
    cleanup='mg_tilejp2_cleanup'
end
```

This has initialized the program; the rest of the routines respond to user actions.

The cleanup routine simply destroys the object graphics hierarchy and frees the *pstate* pointer:

```
pro mg_tilejp2_cleanup, tlb
    compile_opt strictarr

    widget_control, tlb, get_uvalue=pstate

    obj_destroy, (*pstate).oview
    ptr_free, pstate
end
```

The JPEG2000 object used in the program is not kept; it is created and destroyed as needed, so it does not need to be deleted here.

The event handler for the program merely dispatches events to other routines based on the *UNAME* of the widget that generated the event.

```
pro mg_tilejp2_event, event
    compile_opt strictarr

    uname = widget_info(event.id, /uname)

    case uname of
        'tlb' : mg_tilejp2_resize, event
        'draw' : mg_tilejp2_draw, event
    endcase
end
```

This scheme scales well for larger, more complicated programs that have many widgets generating events.

The draw widget event handler must handle press, release, motion, and keyboard events. The *buttonsDown* field of the *state* structure, like *event.press* and *event.release*, is a bitmask containing the buttons currently pressed (1 for left mouse button, 2 for middle, 4 for right). The press event also remembers the location of the mouse if it was a left button press:

```
pro mg_tilejp2_draw, event
    compile_opt strictarr

    widget_control, event.top, get_uvalue=pstate

    case event.type of
        0 : begin                                ; press event
```

```

(*pstate).buttonsDown or= event.press
if (event.press and 1B) then begin
  (*pstate).pressLoc = (*pstate).curLoc + [event.x, event.y] * (*pstate).zoomFactor
endif
end
1 : (*pstate).buttonsDown and= not event.release ; release event

```

When the left mouse button is held and the mouse is moved, a new location for the viewport is calculated by converting event.x and event.y to pixel coordinates and determining the shift from the location where the mouse was originally pressed:

```

2 : begin ; motion event
  if ((*pstate).buttonsDown and 1B) then begin
    loc = (*pstate).pressLoc - [event.x, event.y] * (*pstate).zoomFactor
    mg_tilejp2_move, pstate, loc
  endif
end

```

Expose events just require redraw of the window; scroll and ASCII keyboard events are ignored:

```

3 : ; scroll event
4 : (*pstate).owindow->draw, (*pstate).oview ; expose event
5 : ; ASCII key event

```

Non-ASCII keyboard presses are handled next. The arrow keys move an eighth of the window size in the given direction. Page up/down increase/decrease the zoom level by 1. The home key resets to the original location and zoom level.

```

6 : begin ; non-ASCII key event
  if (event.release ne 0) then return
  winSize = (*pstate).windowDims * (*pstate).zoomFactor
  case event.key of
    5 : begin ; left
      loc = (*pstate).curLoc + [- winSize[0] / 8, 0]
      mg_tilejp2_move, pstate, loc
    end
    6 : begin ; right
      loc = (*pstate).curLoc + [winSize[0] / 8, 0]
      mg_tilejp2_move, pstate, loc
    end
    7 : begin ; up
      loc = (*pstate).curLoc + [0, winSize[1] / 8]
      mg_tilejp2_move, pstate, loc
    end
    8 : begin ; down
      loc = (*pstate).curLoc + [0, - winSize[1] / 8]
      mg_tilejp2_move, pstate, loc
    end
    9 : mg_tilejp2_zoom, pstate, 1 ; page up
    10 : mg_tilejp2_zoom, pstate, -1 ; page down
    11 : begin ; home
      (*pstate).curLoc = [0L, 0L]
      (*pstate).zoomLevel = 0L
      (*pstate).zoomFactor = 2.0^(*pstate).zoomLevel
      mg_tilejp2_setvp, pstate, [(*pstate).curLoc, (*pstate).windowDims]
    end
  else :
  endcase
end

```

For scroll wheel events, the *clicks* field contains negative values for scrolling down and positive values for scrolling up (usually 1 and -1, but they can be higher if the user scrolls quickly). This value is passed to `MG_TILEJP2_ZOOM` to change the zoom level:

```

    7 : begin                                ; wheel event
        mg_tilejp2_zoom, pstate, event.clicks
    end
endcase
end

```

Next, let's tackle the helper routines called by this event handler.

The `MG_TILEJP2_MOVE` procedure does a translation of the viewport to the given location *loc*:

```

pro mg_tilejp2_move, pstate, loc
    compile_opt strictarr

    (*pstate).curLoc = loc
    vp = [loc, (*pstate).windowDims * (*pstate).zoomFactor]
    mg_tilejp2_setvp, pstate, vp
end

```

Calling this routine changes the `VIEWPLANE_RECT` property of the view and redraws the window.

The `MG_TILEJP2_SETVP` procedure is a convenience to set the `VIEWPLANE_RECT` property of the view and redraw the scene.

```

pro mg_tilejp2_setvp, pstate, vp
    compile_opt strictarr

    (*pstate).oview->setProperty, viewplane_rect=vp
    mg_tilejp2_refresh, pstate
end

```

Redrawing the scene is not simply a call to `IDLgrWindow::draw` though; the appropriate tiles need to be read and loaded into the image in the `MG_TILEJP2_REFRESH` procedure before the *draw* method can be called.

The `MG_TILEJP2_REFRESH` procedure queries the window for what tiles are needed, reads the data needed from the JPEG2000 file, loads them into the image, and redraws the window. It's a helper routine that takes a pointer to the *state* structure, getting the image object by name:

```

pro mg_tilejp2_refresh, pstate
    compile_opt strictarr

    oimage = (*pstate).oview->getByName('model/image')

```

Next, the tiles required to display the current viewport are calculated:

```

reqTiles $
    = (*pstate).owindow->queryRequiredTiles((*pstate).oview, $
                                           oimage, $
                                           count=nTiles)

```

Let's turn the mouse into an hourglass if we actually have tiles to read and load:

```

if (nTiles gt 0) then widget_control, /hourglass

```

We could create an `IDLffJPEG2000` object in two places: either outside the loop over the tiles or inside the loop. The *PERSISTENT* keywords needs to be set differently in these two cases. If only one data access is to be done on the object, *PERSISTENT* can be set to 0, which minimizes memory by discarding data whenever possible. If multiple accesses are needed, *PERSISTENT* must be set to 1, the default:


```
objp2 = obj_new('IDLffJPEG2000', (*pstate).jp2filename, $
               persistent=1)
```

Finally, we are ready to pull each tile from the data file and load it into the image object.

```
for t = 0L, nTiles - 1L do begin
    subrect = [reqTiles[t].x, reqTiles[t].y, $
              reqTiles[t].width, reqTiles[t].height]

    level = reqTiles[t].level
    scale = ishft(1, level) ; scale = 2^level
    subrect *= scale

    tileData = objp2->getData(region=subrect, $
                             discard_levels=level, $
                             order=1)

    oimage->setTileData, reqTiles[t], tileData, no_free=0
endfor
```

Note that ISHFT is used to efficiently compute 2^{level} ; it is commonly used to multiply or divide a value by a power of two. The JPEG2000 object is no longer needed and can be destroyed:

```
obj_destroy, objp2
```

Finally, the *draw* method displays the results:

```
(*pstate).owindow->draw, (*pstate).oview
end
```

The hourglass does not need to be turned off—it will automatically revert when the event handler finishes executing.

The `MG_TILEJP2_ZOOM` procedure zooms in or out from the current level. The main computation of the routine is the calculation of a value of the `VIEWPLANE_RECT` property based on the current location and zoom level and modified by the change in the zoom level, *incLevel*.

```
pro mg_tilejp2_zoom, pstate, incLevel
    compile_opt strictarr

    (*pstate).zoomLevel += incLevel

    dims = (*pstate).windowDims * (*pstate).zoomFactor
    (*pstate).zoomFactor = 2.0^(*pstate).zoomLevel
    (*pstate).curLoc += (dims - (*pstate).windowDims * (*pstate).zoomFactor) / 2.0

    vp = [(*pstate).curLoc, (*pstate).windowDims * (*pstate).zoomFactor]
    mg_tilejp2_setvp, pstate, vp
end
```

The `MG_TILEJP2_SETVP` procedure is called to change the `VIEWPLANE_RECT` and refresh the display.

The *resize* routine does straightforward widget resizing of the draw widget, but the content in the draw widget must be updated because more tiles may be needed if the window is larger.

```
pro mg_tilejp2_resize, event
    compile_opt strictarr

    widget_control, event.top, get_uvalue=pstate

    tlbG = widget_info(event.top, /geometry)
    draw = widget_info(event.top, find_by_undef='draw')
```

```

(*pstate).windowDims = [event.x - 2 * tlbG.xpad, event.y - 2 * tlbG.ypad]
widget_control, draw, $
    xsize=(*pstate).windowDims[0], $
    ysize=(*pstate).windowDims[1]

; fix up object graphics hierarchy
vp = [(*pstate).curLoc, (*pstate).windowDims * (*pstate).zoomFactor]
mg_tilejp2_setvp, pstate, vp
end

```

The MG_TILEJP2_SETVP procedure is again called to load new tiles and refresh the display. The dimensions of the draw widget are saved during this calculation because they are needed in the MG_TILEJP2_ZOOM procedure.

The MG_TILEJP2_DEMO procedure is a wrapper around MG_TILEJP2 that loads a default data set. The MG_TILEJP2 procedure uses a JPEG2000 image. Because a JPEG2000 image is not provided with the IDL distribution, one is created from *ohare.jpg*, a 5000 by 5000 pixel TrueColor image included in the examples data.

```

pro mg_tilejp2_demo, renderer=renderer
  compile_opt strictarr

  jp2filename = 'ohare.jp2'

  ; check if ohare.jp2 is present, create if not present
  if (~file_test(jp2filename)) then begin
    jpegFilename = filepath('ohare.jpg', $
                          subdirectory=['examples', 'data'])
    read_jpeg, jpegFilename, jpegImage
    imageDims = size(jpegImage, /dimensions)

    ; create the JPEG2000 image object
    jp2 = obj_new('IDLffJPEG2000', jp2filename, write=1)
    jp2->setProperty, n_components=3, $
                  n_layers=20, $
                  n_levels=6, $
                  offset=[0, 0], $
                  tile_dimensions=[1024, 1024], $
                  tile_offset=[0, 0], $
                  bit_depth=[8, 8, 8], $
                  dimensions=[imageDims[1], imageDims[2]]

    ; Set image data, and then destroy the object. You must create
    ; and close the JPEG2000 file object before you can access the
    ; data.
    jp2->setData, jpegImage
    obj_destroy, jp2
  endif

  ; start mg_tilejp2
  mg_tilejp2, jp2filename, renderer=renderer
end

```

The *IDLffJPEG2000::setProperty* call sets the image object to use 3 channels (with 8 bits per channel), 20 quality layers, 6 wavelet decomposition levels, and 1024 by 1024 pixel tiles.

Exercises

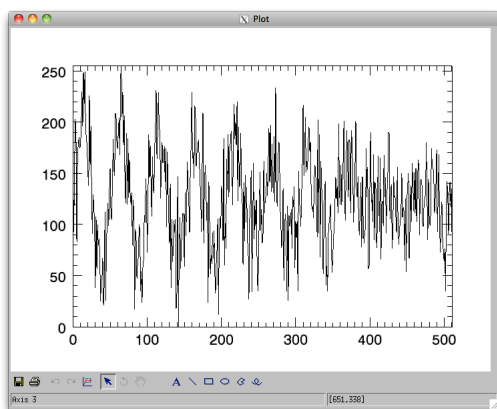
The following exercises describe modifications to the MG_TILEJP2 program.

1. Export the visible portion of the image to a PNG file when the “s” key is pressed.

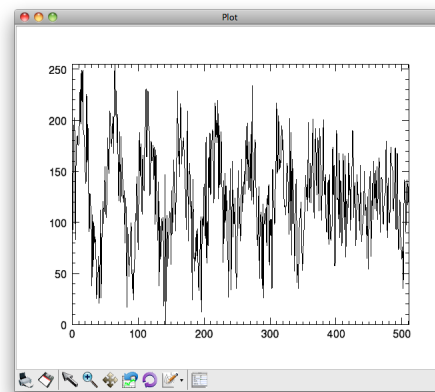
9.15. Function graphics

Introduced in IDL 8.0, “function graphics”¹ provide an easy-to-use interface to object graphics without the low-level details of the object graphics system or the complexity of the iTools system. Although function graphics are based on the iTools framework, they have an entirely different interface.

Table 9.9. Examples of function graphics displays



Display of example data file using function graphics from the command line uses the default widget system (shown for Mac OS X).



Display of example data file using function graphics from the IDL Workbench uses the Qt widget toolkit.

The function graphics interface combines much of the ease of use of direct graphics with the power and flexibility of object graphics.

Let’s do a simple example to demonstrate the simple function graphics interface. First, read in two 1-dimensional data sets to plot:

```
IDL> waves = (read_ascii(file_which('sine_waves.txt'))).field1
```

Create the plot object, along with its corresponding display, with the data from the first curve:

```
IDL> p1 = plot(waves[0, *])
```

The *p1* variable above is an object and can be used later to refer to the plot, change properties of the plot, or perform other actions on the plot:

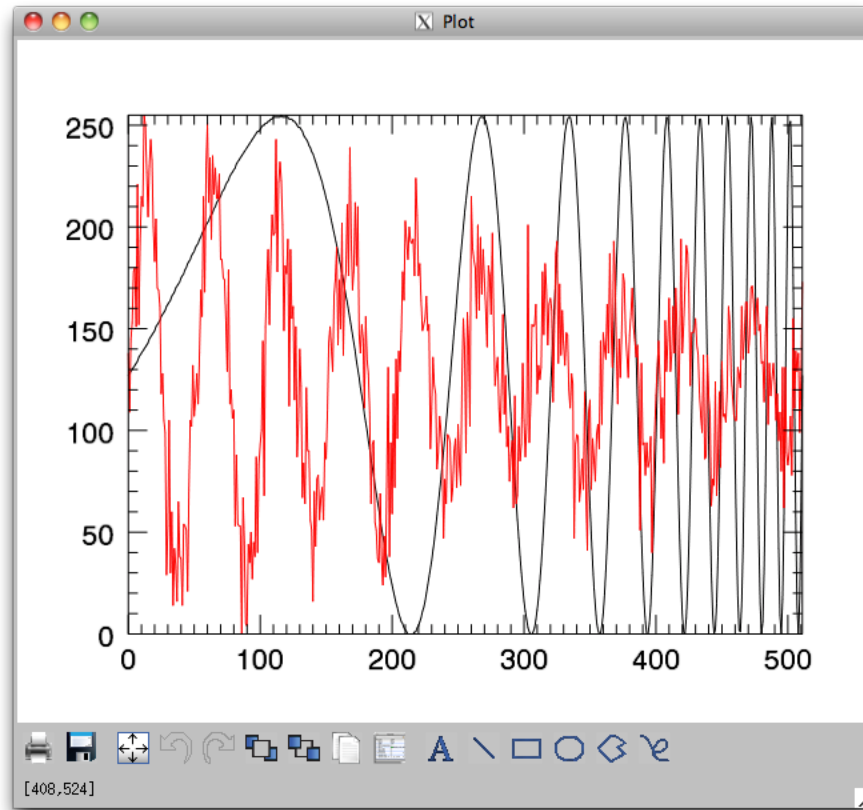
```
IDL> help, p1
P1          PLOT <21286>
```

The second data set can be overplotted on the first plot using the *OVERPLOT* keyword:

```
IDL> p2 = plot(waves[1, *], /overplot, color='red')
```

¹Function graphics were introduced as “graphics” by ITT VIS, but that leads to much confusion in practice. The `comp.lang.idl-pwave` newsgroup calls them “new graphics” or “function graphics”, so the “function graphics” convention will be used in this book.

This should produce the following display:



Simple function graphics display of two vector data sets as overplotted line plots.

Properties of the plot can be changed easily using the dot notation:

```
IDL> p1.linestyle = 'dashed'
```

The available properties for a function graphics line plot can be found in the online help for the `PLOT` function, but they can also be listed, along with their values, just by printing the object:

```
IDL> print, p1
PLOT <28162>
  ANTIALIAS           = 1
  ASPECT_RATIO        = 0.0000000
  ASPECT_Z            = 0.0000000
  BACKGROUND_COLOR    = 255 255 255
  COLOR               = 0 0 0
  DEPTH_CUE           = 0.00000 0.00000
  ERRORBAR_CAPSIZE     = 0.20000000
  ERRORBAR_COLOR       = 0 0 0
  FILL_BACKGROUND      = 0
  FILL_COLOR           = 128 128 128
  FILL_LEVEL           = 1.0000000e-300
  FILL_TRANSPARENCY    = 0
  HIDE                 = 0
  LINESYLE             = 2
  MAX_VALUE            = NaN
  MIN_VALUE            = NaN
```

```

NAME           = 'Plot'
SYMBOL         = 0
SYM_COLOR      = 0  0  0
SYM_FILLED     = 0
SYM_FILL_COLOR = 0  0  0
SYM_INCREMENT  = 1
SYM_SIZE       = 1.0000000
SYM_THICK      = 1.00000
SYM_TRANSPARENCY = 0
THICK          = 1.00000
TITLE          = <NullObject>
TRANSPARENCY   = 0
WINDOW_TITLE   = 'Plot'
XRANGE         = 0.0000000  511.00000
YRANGE         = 0.0000000  255.00000
ZRANGE         = 0.0000000  0.0000000

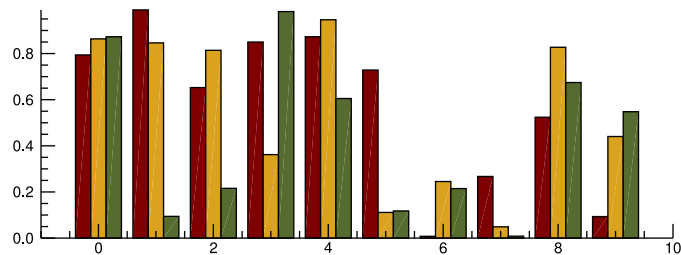
```

These properties can be accessed using the dot notation or the standard *getProperty/setProperty* methods.

The following table gives a short description and some examples of the function graphics routines.

Table 9.10. Function graphics routines

Routine	Description
BARPLOT	<p>Syntax:</p> <pre> graphic = barplot(values [, _extra=e]) graphic = barplot(locations, values [, _extra=e]) </pre> <p>The BARPLOT is used to create various types of bar plots, including stacked bar plots, grouped bar plots, and floating bar plots. For example, the following should produce a grouped bar plot of ten series of three bars:</p> <pre> nBars = 3 nSeries = 10 colors = ['maroon', 'goldenrod', 'dark_olive_green'] data = randomu(seed, nSeries, nBars) b = barplot(data[, 0], nbars=nBars, index=0, fill_color=colors[0], \$ axis_style=1, font_size=11, dimensions=[800, 300]) for i = 1L, nBars - 1L do \$ b = barplot(data[, i], nbars=nBars, index=i, fill_color=colors[i], \$ axis_style=1, /overplot, font_size=11) </pre> <p>The graphic should look like:</p>

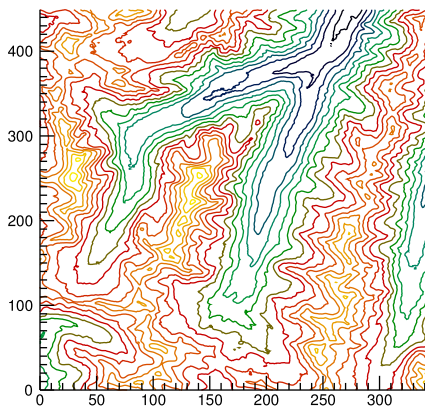


CONTOUR

Syntax:

```
graphic = contour(z [, x, y] [, _extra=e])
```

Routine	Description
	<p>The CONTOUR procedure is used to create a contour plot. For example, a standard contour plot can be created easily:</p> <pre>restore, filename=file_which('marbells.dat') fill = contour(elev, n_levels=20, rgb_table=4, dimensions=[500, 500], \$ font_size=11)</pre> <p>This should produce the following graphic:</p>



There are options to create filled contours and 3-dimensional contour plots as well.

ERRORPLOT

Syntax:

```
graphic = errorplot(y, yerror [, format], [, _extra=e])
graphic = errorplot(x, y, xerror, yerror [, format] [, _extra=e])
graphic = errorplot(x, y, yerror [, format] [, _extra=e])
```

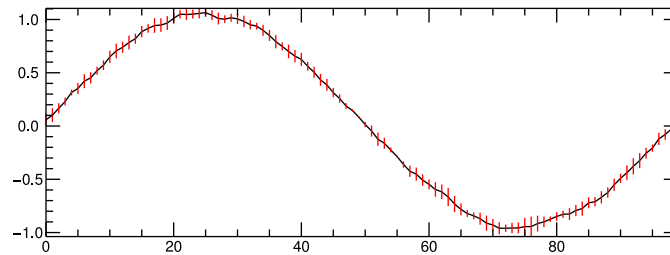
The ERRORPLOT function produces a line plot with error bands. It can accept error values for the y-axis values or both the x- and y-axis values. For example, the following constructs a simple example:

```
n = 100
y = sin(findgen(n) / (n - 1) * 360 * !dtr) $
    + smooth(randomu(seed, 360) * 0.1, 3, /edge_truncate)
yerror = randomu(seed, n) * 0.1
yerror = smooth(yerror, 3, /edge_truncate)
x = findgen(n)
p = errorplot(x, y, yerror, errorbar_capsize=0., errorbar_color='red', $
              dimensions=[800, 300], font_size=11)
```

The graphic should look like:

Routine

Description



IMAGE

Syntax:

```
graphic = image(data [, x, y] [, _extra=e])
graphic = image(filename)
```

Images can be displayed given a filename to the image data in a common image file format:

```
IDL> im = image(file_which('people.jpg'))
```

Image data can also be displayed directly:

```
IDL> data = read_image(file_which('endocell.jpg'))
IDL> im = image(data, findgen(615), findgen(416), rgb_table=5, axis_style=2)
```

There are many properties to an image object which control aspects of the image and axis display. Here, the *RGB_TABLE* keyword is specifying to use color table 5 (equivalent to using *loadct*, 5 in direct graphics). The *AXIS_STYLE* keyword indicates that box axes should be placed around the image: 0 is no axis, the default; 1 is axes on the minimum edge; 2 is box axes; and 3 is crosshair axes.

MAP

Syntax:

```
graphic = map(projection [, _extra=e])
```

The *MAP* function initializes a map projection for a particular area. Further data and annotations can be made with map-specific function graphics routines, like *MAPGRID* or *MAPCONTINENTS*, or other general function graphics annotation routines. For example, the following code sets up a map and displays some annotations:

```
map = map('Mercator', limit=[32, -120, 46, -92], $
         fill_color='light_blue', title='Colorado')

grid = map.mapgrid
grid.linestyle = 'dotted'
grid.label_position = 0
grid.font_size = 11
grid.grid_latitude = 2
grid.grid_longitude = 2

states = mapcontinents(/usa, fill_color='light green', thick=2., combine=0)

m = map['Colorado']
m.fill_color = 'orange'

map['40N'].linestyle = 'dashed'

area = mapgrid(color='gray', $
               longitude_min=-112., longitude_max=-104.5, $
               latitude_min=35.5, latitude_max=41.5, $
               grid_longitude=0.5, grid_latitude=0.5, $
```

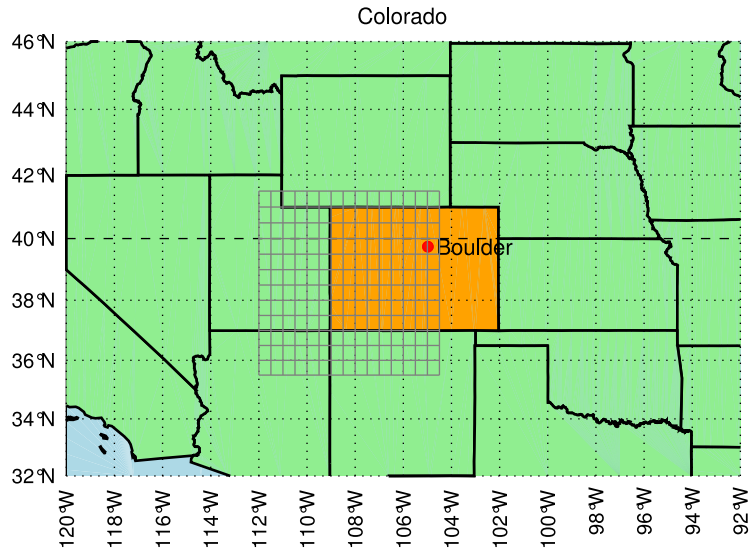
Routine**Description**

```
label_show=0)
```

```
; the SYMBOL routine was added in IDL 8.1
```

```
sym = symbol(-104.98, 39.74, 'circle', /sym_filled, $  
            sym_color='red', label_string='Boulder', /data)
```

This should produce the following graphic:

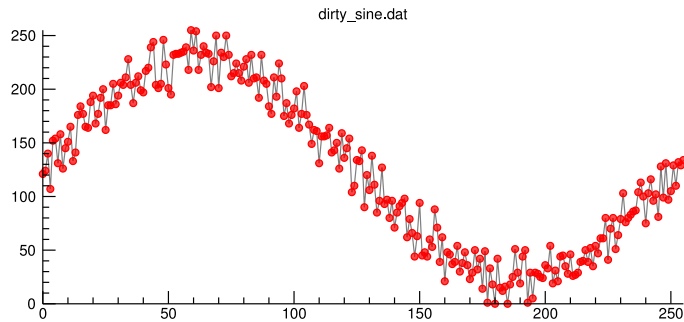
**PLOT****Syntax:**

```
graphic = plot(y [, format] [, _extra=e])  
graphic = plot(x, y [, format] [, _extra=e])
```

The PLOT function produces scatter and line plots, similar to the old PLOT direct graphics routine. For example, the following produces a line plot with the data points marked by symbols:

```
d = read_binary(file_which('dirty_sine.dat'), type=1, data_dims=[256])  
p = plot(d, color='grey', title='dirty_sine.dat', axis_style=1, $  
        symbol='circle', sym_color='red', /sym_filled, sym_size=0.75, $  
        dimensions=[900, 400])
```

The output should look like the plot below.

Routine**Description**

The possible symbol values are listed below, with most symbol types having a case-insensitive standard name and a case-sensitive short form:

“None” (the default)	“Less_than” or “<”	“Pentagon” or “p”
“Plus” or “+”	“Triangle_down” or “td”	“Hexagon_1” or “h”
“Asterisk” or “*”	“Triangle_left” or “tl”	“Hexagon_2” or “H”
“Period” or “dot”	“Triangle_right” or “tr”	“Vline” or “ ”
“Diamond” or “D”	“Tri_up” or “Tu”	“Hline” or “_”
“Triangle” or “tu”	“Tri_down” or “Td”	“Star” or “S”
“Square” or “s”	“Tri_left” or “Tl”	“Circle” or “o”
“X”	“Tri_right” or “Tr”	
“Greater_than” or “>”	“Thin_diamond” or “d”	

PLOT3D

Syntax:

```
graphic = plot3d(x, y, z [, format] [, _extra=e])
```

The PLOT3D function creates 3-dimensional scatter or line plots. The following code produces a spiral line plot:

```
nturns = 3
t = findgen(360 * nturns) * !dtr
x = cos(t)
y = sin(t)
z = t
p = plot3d(x, y, z, color='red', thick=2., $
    axis_style=2, $
    xy_shadow=1, yz_shadow=1, xz_shadow=1, shadow_color='light blue')
axes = p.axes
axes[2].hide = 1
axes[6].hide = 1
axes[7].hide = 1
```

POLARPLOT

Syntax:

```
graphic = polarplot(theta [, format] [, _extra=e])
graphic = polarplot(r, theta [, format] [, _extra=e])
```

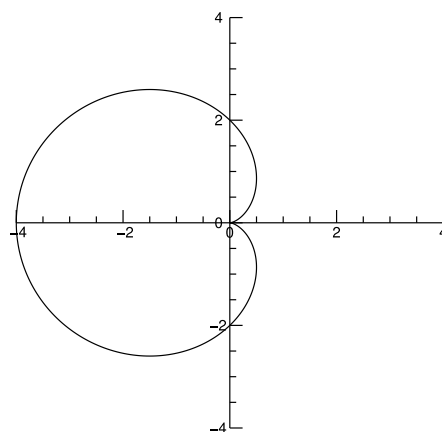
The POLARPLOT graphics function produces polar plots, i.e., line plots specified in polar coordinates like the following plot of a cardioid:

```
theta = findgen(360) * !dtr
r = 2. * (1. - cos(theta))
p = polarplot(r, theta, axis_style=3, xrange=[-4, 4], yrange=[-4, 4])
```

Routine

Description

This produces the following plot:



STREAMLINE

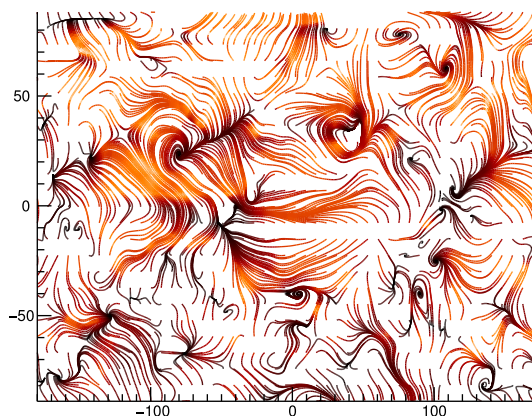
Syntax:

```
graphic = streamline(u, v [, x, y] [, format] [, _extra=e])
```

The STREAMLINE function draws streamlines from a grid of initial points:

```
restore, filepath('globalwinds.dat', subdir=['examples', 'data'])
s = streamline(u, v, x, y, rgb_table=3, auto_color=1, $
    x_streamparticles=50, y_streamparticles=50, arrow_size=0., $
    font_size=11)
```

This should produce the following graphic:

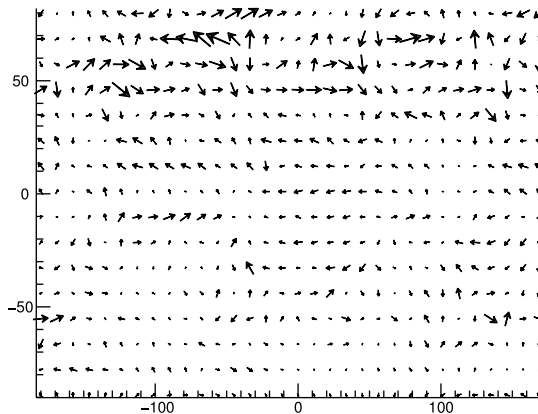


SURFACE

Syntax:

```
graphic = surface(z [, x, y] [, format] [, _extra=e])
```

Routine	Description
	<p>The SURFACE graphics function can produce surface plots of 2-dimensional data. It can also drape imagery over the surface as in this code:</p> <pre>elev = read_binary(file_which('elevbin.dat'), data_dims=[64, 64], type=1) image = read_image(file_which('elev_t.jpg')) s = surface(elev, texture_image=image, zrange=[0, 1000])</pre>
VECTOR	<p>Syntax:</p> <pre>graphic = vector(u, v [, x, y] [, format] [, _extra=e])</pre> <p>The VECTOR function creates a standard grid of arrows or wind barbs to visualize a vector field. The 2-dimensional components of the vector field are passed to VECTOR first, then, optionally, the values for the <i>x</i>- and <i>y</i>-axes. There are multiple keywords to control aspects of the arrows. For example, the following creates a basic vector field plot:</p> <pre>restore, filepath('globalwinds.dat', subdir='examples', 'data') vec = vector(u[0:*:4], v[0:*:4], x[0:*:4], y[0:*:4], \$ /head_proportional, head_size=0.5, length_scale=0.5, \$ font_size=11)</pre> <p>This should produce the following graphic:</p>



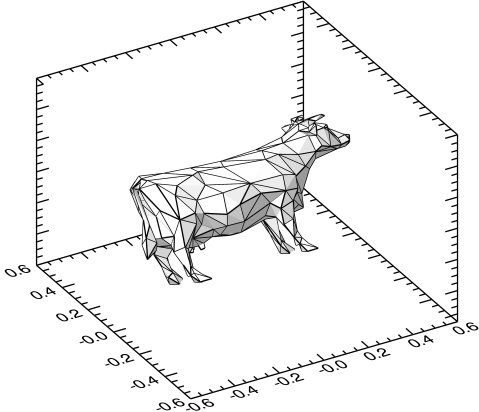
All function graphics objects have *close*, *convertCoord*, *copyWindow*, *getSelect*, *order*, *print*, *refresh*, *rotate*, *save*, *scale*, *select*, and *translate* methods.

The following is a list of helper routines used for producing annotations on existing graphics with the exception of the WINDOW function to create function graphics windows.

Table 9.11. Function graphics helper routines

Routine	Description
ARROW	<p>Syntax:</p> <pre>graphic = arrow(x, y [, z] [, _extra=e])</pre> <p>The ARROW function creates arrow annotations:</p> <pre>p = plot(/test) a = arrow([150., 132.], [0.5, 0.3], /data, head_size=0.5)</pre>

Routine	Description
AXIS	<pre data-bbox="456 233 974 258">t = text(151., 0.5, 'Important point', /data)</pre> <hr/> <p data-bbox="440 279 513 300">Syntax:</p> <pre data-bbox="456 321 893 346">graphic = axis(direction [, _extra=e])</pre> <p data-bbox="440 373 1396 499">The <code>AXIS</code> function creates a new axis on the current graphic window. The positional parameter specifies the direction of the axis: “x”, “y”, or “z”. The <i>LOCATION</i> keyword sets the location of the axis, while the <i>TEXTPOS</i> and <i>TICKDIR</i> determine on which side of the axis the labels and tickmarks appear. The following code places an inches and a centimeter scale on a plot of values:</p> <pre data-bbox="456 520 1226 722">n = 100 data = smooth(randomu(seed, n), 3, /edge_truncate) p = plot(data, margin=[0.15, 0.15, 0.15, 0.15], \$ axis_style=1, thick=2., yrange=[0., 1.], ytitle='inches') yaxis = axis('y', location=[n - 1., 0.], \$ title='cm', tickdir=1, textpos=1, ticklen=0.025, \$ tickvalues=findgen(6) / 2. / 2.54, \$ tickname=string(findgen(6) / 2., format='(%%0.1f)'))</pre>
COLORBAR	<p data-bbox="440 743 513 764">Syntax:</p> <pre data-bbox="456 785 802 810">graphic = colorbar([_extra=e])</pre> <p data-bbox="440 837 1406 932">Used to create a colorbar. The <i>TARGET</i> property specifies a graphics object whose color table will be used to construct the colorbar. Set <i>ORIENTATION</i> to 0 for horizontal or 1 for vertical. For example, try:</p> <pre data-bbox="456 953 1308 1073">restore, filename=file_which('marbells.dat') im = image(elev, rgb_table=9, position=[0.25, 0.05, 0.95, 0.95], \$ dimensions=[500, 500]) c = colorbar(target=im, orientation=1, position=[0.15, 0.05, 0.2, 0.95], \$ font_size=11, ticklen=0.2)</pre>
ELLIPSE	<p data-bbox="440 1094 513 1115">Syntax:</p> <pre data-bbox="456 1136 1446 1161">graphic = ellipse(x, y, [, z] [, format] [, major=major] [, minor=minor] [, _extra=e])</pre> <p data-bbox="440 1188 1122 1213">The <code>ELLIPSE</code> function draws an ellipse on an existing graphic window:</p> <pre data-bbox="456 1234 984 1278">p = plot(findgen(11), /nodata) e = ellipse(5., 5., major=2., minor=1., /data)</pre>
LEGEND	<p data-bbox="440 1299 513 1320">Syntax:</p> <pre data-bbox="456 1341 802 1367">graphic = legend([_extra=e])</pre> <p data-bbox="440 1394 1419 1488">Used to create a legend. Use the <i>TARGET</i> property to specify graphics objects to be listed in the legend. Properties of the specified graphics are used in the listing, particularly the <i>NAME</i> property. For example, try:</p> <pre data-bbox="456 1509 1357 1709">restore, filename=file_which('plot_data.sav') help, plot_data, /structures p1 = plot(plot_data.time, plot_data.temp1, name='Temp 1', color='coral', \$ thick=2, dimensions=[700, 300], font_size=12) p2 = plot(plot_data.time, plot_data.temp2, name='Temp 2', /overplot, \$ color='cadet_blue', thick=2) legend = legend(target=[p1, p2], position=[0.2, 0.80], /normal, linestyle=6, \$ font_size=12)</pre>
MAPCONTINENTS	<p data-bbox="440 1730 513 1751">Syntax:</p> <pre data-bbox="456 1772 1008 1797">graphic = mapcontinents([filename] [, _extra=e])</pre>

Routine	Description
	<p>The MAPCONTINENTS function places continent outlines, country outlines, US state outlines, Canada province outlines, lakes, or rivers on a map. See the description of the MAP function above for an example of using MAPCONTINENTS.</p>
MAPGRID	<p>Syntax:</p> <pre>graphic = mapgrid([, _extra=e])</pre> <p>The MAPGRID function places gridlines on a map. See the description of the MAP function above for an example of using MAPGRID.</p>
POLYGON	<p>Syntax:</p> <pre>graphic = polygon(x, y [, z] [, _extra=e])</pre> <p>The POLYGON function creates a 2- or 3-dimensional polygon in an existing graphic. For example, the following example code produces the familiar cow polygon:</p> <pre>restore, filename=file_which('cow10.sav') range = [-0.6, 0.6] p = plot3d(range, range, range, /nodata, axis_style=2) axes = p.axes axes[2].hide = 1 axes[6].hide = 1 axes[7].hide = 1 cow = polygon(x, y, z, connectivity=polylist, /data)</pre> <p>This produces the following graphic:</p> 
POLYLINE	<p>Syntax:</p> <pre>graphic = polyline(x, y [, z] [, format] [, _extra=e])</pre> <p>The POLYLINE graphics function draws line segments on an existing plot. For example, the following code shows how to draw a vertical line through a plot:</p> <pre>d = read_binary(file_which('dirty_sine.dat'), type=1, data_dims=[256]) p = plot(d, color='grey', dimensions=[900, 400]) line = polyline(fltarr(2) + 200., p.yrange, color='red', linestyle='dashed', thick=2., /data)</pre>
SYMBOL	<p>Syntax:</p> <pre>symbol = symbol(x, y, symbol)</pre>

Routine	Description
TEXT	<p data-bbox="440 239 948 260">The SYMBOL function places a symbol on a graphic:</p> <pre data-bbox="459 281 1214 331">p = plot(sin(findgen(360) * !dtr)) s = symbol(0., 0., 'star', sym_color='red', label='Origin', /data)</pre> <p data-bbox="440 348 1365 407">Coordinates can be specified in data, normal (the default), and device coordinates. The SYMBOL function was added in IDL 8.1.</p>
WINDOW	<p data-bbox="440 428 513 449">Syntax:</p> <pre data-bbox="459 470 1110 491">graphic = text(x, y, [z,] string [, format] [, _extra=e])</pre> <p data-bbox="440 520 862 541">The TEXT function places text on a graphic:</p> <pre data-bbox="459 562 1089 642">p = plot(sin(findgen(360) * !dtr)) t = text(0., 0., 'Origin', /data) t = text(0.5, 0.925, 'Heading', alignment=0.5, /normal)</pre> <p data-bbox="440 659 1224 680">Coordinates can be specified in data, normal, and device coordinates (the default).</p>
	<p data-bbox="440 701 513 722">Syntax:</p> <pre data-bbox="459 743 802 764">graphic = window([, _extra=e])</pre> <p data-bbox="440 793 1398 919">An empty graphics window can be created with the WINDOW function. Use the <i>CURRENT</i> keyword to make the other graphics window routines that would normally create their own window to instead use the current graphics window. For example, the following creates a window separately from the line plot visualization:</p> <pre data-bbox="459 940 834 991">w = window(dimensions=[800, 300]) p = plot(findgen(11), /current)</pre> <p data-bbox="440 1008 1105 1029">A WINDOW object has <i>close</i>, <i>convertCoord</i>, <i>print</i>, and <i>save</i> methods.</p> <p data-bbox="440 1066 1247 1087">The currently available windows can be obtained from the GETWINDOWS function:</p> <pre data-bbox="459 1108 1149 1209">IDL> p = plot(/test) IDL> c = contour(/test) IDL> print, getwindows() <ObjHeapVar15493 (GRAPHICSWIN)><ObjHeapVar22443 (GRAPHICSWIN)></pre> <p data-bbox="440 1226 1328 1285">Buffers are not included in this, only displayed windows. The GETWINDOWS also accepts a parameter for the name of the window to retrieve. The first match will be returned.</p> <p data-bbox="440 1318 1409 1377">To create a function graphics window in a widget program, use WIDGET_WINDOW. There is a short example program MG_WIDGET_WINDOW_EXAMPLE described below.</p>

It is possible to create graphics which are not displayed on the screen, a useful technique when creating many graphics output files like PNG or Postscript files. Use the *BUFFER* keyword to create a graphic which isn't displayed:

```
IDL> p = plot(sin(findgen(360) * !dtr), /buffer)
```

The *save* method can then be used to send the output from the graphic to an output file:

```
IDL> p->save, 'sinewave.png', width=600
```

The *WIDTH* keyword sets the width of the PNG file in pixels (in inches or centimeters for PDF output); the height will be calculated from the aspect ratio of the graphic. The raw screen buffer data can be obtained from a graphic using:

```
IDL> im = p.window.image_data
IDL> help, im
IM          BYTE          = Array[3, 640, 512]
IDL> window, xsize=640, ysize=512
IDL> tv, im, true=1
```

Note that displayed graphics are automatically cleaned up when their windows are closed, but graphics displayed in buffers should be manually destroyed:

```
IDL> p->close
```

Of course, the automatic garbage collection in IDL 8.0 would automatically free the buffer graphic when there are no more object references to it.

It is also possible to temporarily stop a window from refreshing in order to make several changes. This improves the speed of the drawing as well as eliminating any flashing when the redrawing is done. For example, the example for the `BARPLOT` function above drew its output in several calls to `BARPLOT` with the `OVERPLOT` keyword set, each redraw adding a new set of bars and possibly changing the scale of the draw. In the below modification, we turn off the redrawing in the window until the graphic is complete:

```
nBars = 3
nSeries = 10
colors = ['maroon', 'goldenrod', 'dark_olive_green']
data = randomu(seed, nSeries, nBars)
w = window(dimensions=[800, 300])
w->refresh, /disable
b = barplot(data[*, 0], nbars=nBars, index=0, fill_color=colors[0], $
            axis_style=1, /current, font_size=11)
for i = 1L, nBars - 1L do $
    b = barplot(data[*, i], nbars=nBars, index=i, fill_color=colors[i], $
                axis_style=1, /overplot, font_size=11)
w->refresh
```

The explicit creation of the graphics window and the two calls to `window::refresh`, one to disable refreshing and one to turn it back on, are the only differences from the previous example.

There are a few methods that are common for all the graphic classes. Listed below are the common methods and their most common keywords.

```
graphic::close
```

The *close* method closes the graphic's window.

```
graphic::delete
```

The *delete* method deletes the graphic from the window it was displayed in. Added in IDL 8.1.

```
im = graphic::copyWindow([HEIGHT=h] [, WIDTH=w] [, RESOLUTION=dpi] [, TRANSPARENT=rgb] ...)
```

The *copyWindow* method returns a snapshot of the graphic's window.

```
graphic::save [, HEIGHT=h] [, WIDTH=w] [, RESOLUTION=dpi] ...
```

The *save* method saves the contents of the graphics's window in an image file.

```
graphic::getData, arg1, arg2, arg3, ... [, CONNECTIVITY=conn]
```

The *getData* method gets the data associated with the graphic. The number of parameters is based on the number of arguments used when creating the graphic. Added in IDL 8.1.

```
graphic::setData, arg1, arg2, arg3, ... [, CONNECTIVITY=conn]
```

The *setData* method sets the data associated with the graphic. The number of parameters is based on the number of arguments used when creating the graphic. Added in IDL 8.1.

```
coords = graphic::convertCoord(x [, y] [, z] [/DATA] [, /NORMAL] [, /DEVICE]
                               [/TO_DATA] [, /TO_NORMAL] [, /TO_DEVICE])
```

The *convertCoord* method converts between data, normal, and device coordinates in the graphic.

```
value = graphic::getValueAtLocation(x [, y] [, z] [, /DEVICE] [, /NORMAL] ...)
```

The *getValueAtLocation* method returns the closest data value to the specified point. The input coordinates are assumed to be in data coordinates unless *DEVICE* or *NORMAL* are set. Added in IDL 8.1.

```
xy = graphic::mapForward(lon [, lat] [, /RADIANS] ... )
```

The *mapForward* method transforms latitude/longitude coordinates to (x,y) coordinates.

```
lon_lat = graphic::mapInverse(x [, y] [, /RADIANS])
```

The *mapInverse* method transforms (x,y) coordinates to latitude/longitude coordinates.

```
graphic::order [, /BRING_FORWARD] [, /BRING_TO_FRONT] [, /SEND_BACKWARD] [, /SEND_TO_BACK]
```

The *order* method controls the relative Z-order of graphics objects.

```
graphic::select
```

The *select* method selects the graphic in its window and brings the window to the front.

```
graphic::refresh [, /REFRESH]
```

The *refresh* method enables and disables the refreshing of the graphics window.

```
graphic::scale, x, y, z
```

The *scale* method scales the graphic item in the x, y, and z dimensions.

```
graphic::rotate [, angle] [, /DEFAULT] [, /RESET] [, /XAXIS] [, /YAXIS] [, /ZAXIS]
```

The *rotate* method rotates the graphic. It can reset the graphic's orientation, set it to a default orientation, or rotate the graphic the specified number of degrees around either the x-, y-, or z-axis. The default orientation is 30 degrees around the x-axis, 30 degrees around the y-axis, and -90 degrees around the z-axis.

```
graphic::translate [, x, y, z] [, /DATA] [, /DEVICE] [, /NORMAL] [, /RESET]
```

The *translate* method translates the graphic. It can reset the translation of the graphic or translate the graphic the specified direction in data, device (the default), or normal coordinates.

See the online help for further details of these methods.

The `MG_WIDGET_WINDOW_EXAMPLE` widget program is an example of putting a function graphics plot into a widget program. The widget creation routine for it uses the `WIDGET_WINDOW` function to create the equivalent of a draw widget for function graphics plots. After the window widget is created and realized, then setting the *CURRENT* keyword in the `PLOT` function call sends the output to the newly created window widget:

```
pro mg_widget_window_example
  compile_opt strictarr

  tlb = widget_base(/column, $
    title='Example of using function graphics in a widget program')

  toolbar = widget_base(tlb, /row)

  linestyles = ['solid', 'dot', 'dash', 'dash dot', 'dash dot dot dot', $
    'long dash', 'none']
  linestyleDrop = widget_droplist(toolbar, value=linestyles, unname='linestyle')

  colors = strlower(tag_names(!color))
  colorsDrop = widget_droplist(toolbar, value=colors, unname='color')
  widget_control, colorsDrop, set_droplist_select=7 ; black

  graphic = widget_window(tlb, xsize=600, ysize=300, unname='graphic', $
    /button_events, /keyboard_events, /motion_events)
```



```

statusbar = widget_text(tlb, scr_xsize=600, uname='statusbar')

widget_control, tlb, /realize

; we don't need to get the function graphics window reference, but it can be
; obtained as usual if needed
widget_control, graphic, get_value=win

; create plot in function graphics window
p = plot(sin(findgen(360) * !dior), /current, thick=2., font_size=11)

state = { tlb: tlb, plot: p }
pstate = ptr_new(state, /no_copy)
widget_control, tlb, set_uvalue=pstate

xmanager, 'mg_widget_window_example', tlb, /no_block, $
      cleanup='mg_widget_window_example_cleanup', $
      event_handler='mg_widget_window_example_event'
end

```

Note, our demo has to explicitly turn on the types of events the WIDGET_WINDOW should generate, in the same manner as for WIDGET_DRAW. The above example demonstrates how the graphics window of the plot can be retrieved with the *GET_VALUE* keyword of WIDGET_CONTROL in the usual manner, though it is not used. The object reference for the plot itself is more useful, in general, so it is saved in the *state* structure.

The cleanup routine for our example only needs to free the *pstate* pointer; the plot object does not need to be freed explicitly, it will be freed when its window is destroyed.

```

pro mg_widget_window_example_cleanup, tlb
  compile_opt strictarr

  widget_control, tlb, get_uvalue=pstate

  ptr_free, pstate
end

```

Our demo generates events from the linestyle and color droplists, as well as the function graphic. The linestyle and color events are used to modify the properties of the plot. The various mouse motion, mouse press/release, and keyboard events from the WIDGET_WINDOW are identified by a message displayed in the status label.

```

pro mg_widget_window_example_event, event
  compile_opt strictarr
  on_error, 2

  widget_control, event.top, get_uvalue=pstate
  uname = widget_info(event.id, /uname)

  case uname of
    'linestyle': (*pstate).plot.linestyle = event.index
    'color': (*pstate).plot.color = !color.(event.index)
    'graphic': begin
      statusbar = widget_info(event.top, find_by_uname='statusbar')
      msg = ''
      case event.type of
        0: msg = string(event.x, event.y, format=('%"Button press at %d, %d"'))
        1: msg = string(event.x, event.y, format=('%"Button release at %d, %d"'))
        2: msg = string(event.x, event.y, format=('%"Mouse moved to %d, %d"'))
        5: msg = string(string(event.ch), format=('%"Key pressed: %s"'))
      end
    end
  end

```

```

6: begin
    keys = ['none', 'shift', 'control', 'caps lock', 'alt', $
           'left', 'right', 'up', 'down', 'page up', $
           'page down', 'home', 'end']
    msg = string(keys[event.key], format='(%s"Key pressed: %s")')
    end
endcase

    widget_control, statusbar, set_value=msg
end
else: message, 'unknown widget generating events'
endcase
end

```

See *mg_widget_window_example.pro* in the example code for complete code listing.

For some time, there has been a third-party tool called TEXTOIDL which converted strings using a subset of the syntax from the popular TeX typesetting system to IDL’s graphics format codes. In IDL 8.0, this capability is now built-in—just enclose the TeX mathematics formatting in \$ signs like you would in TeX:

```

IDL> w = window()
IDL> t = text(0.5, 0.5, 'The formula is: $e^{i\pi} + 1 = 0$', font_size=24, alignment=0.5, /normal)

```

But this can be used anywhere in IDL with the TEX2IDL function (note that “to” is replaced by “2” in the ITT VIS version), like:

```

IDL> print, tex2idl('The formula is: $e^{i\pi} + 1 = 0$')
The formula is: e!Ui!Mp!N + 1 = 0
IDL> xyouts, 0.5, 0.5, tex2idl('The formula is: $e^{i\pi} + 1 = 0$'), /normal, alignment=0.5

```

For details on the syntax accepted, see the “Adding Mathematical Symbols and Greek Letters to the Text String” section at the end of the online help for the TEXT function graphics function.

9.16. Summary

1. Object graphics is an object-oriented interface to OpenGL. If the rendering support of a graphics card is not good, object graphics can use the Mesa 3D Graphics Library to emulate OpenGL in software.
2. Setting the *VIEWPLANE_RECT* property on a *IDLgrView* containing an image is the easiest way to control the scaling of an image.
3. Use the *[XYZ]COORD_CONV* properties of graphics items to scale 3-dimensional items into the view volume.
4. Objects are drawn from back to front so add the back items first if the front items are transparent.
5. Object graphics combine well with widget programs. Setup is not so onerous since the widget hierarchy must be initialized as well and the persistent nature of the object graphics hierarchy matches well with interactive nature of widget programs.
6. Set a *WIDGET_DRAW*’s *GRAPHICS_LEVEL* keyword to 2 to display object graphics in the draw widget. Then the value of the draw widget returned from the *GET_VALUE* keyword to *WIDGET_CONTROL* will be an *IDLgrWindow* object.
7. In IDL 8.0 and later, use function graphics as a convenient interface to object graphics.

References

Currently, [PowerGraphics] is the only IDL-specific third party book about object graphics in IDL (although [Primer] is a quick reference guide with an object graphics section). [RedBook] is the definitive reference for OpenGL, the basis for object graphics. [InfoVis] discusses interactive visualizations similar to combining object graphics with widget programs.

[PowerGraphics] Ronn Kling. *Power Graphics with IDL: A Beginners Guide to IDL Object Graphics*. August 2002. KRS, Inc..

Description of object graphics classes and their uses with examples.

[Primer] Ronn Kling. *IDL Primer*. May 14, 2007. Kling Research and Software, Inc.

Quick reference guide for IDL topics including a section on object graphics.

[RedBook] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1*. August 9, 2007. Sixth edition. Addison-Wesley Professional.

Object graphics is just an interface to OpenGL and the “Red Book” is the definitive reference for OpenGL programming.

[InfoVis] Colin Ware. *Information Visualization: Perception for Design (Interactive Technologies)*. April 7, 2004. Second edition. Morgan Kaufmann.

Classic in the field of Information Visualization, the use of interactive computer-based visualizations.

