Object Programming

IDL Version 7.0
November 2007 Edition

1107IDL70OBJ

Restricted Rights Notice

The IDL®, IDL Analyst™, ENVI®, and ENVI Zoom™ software programs and the accompanying procedures, functions, and documentation described herein are sold under license agreement. Their use, duplication, and disclosure are subject to the restrictions stated in the license agreement. ITT Visual Information Solutions reserves the right to make changes to this document at any time and without notice.

Limitation of Warranty

ITT Visual Information Solutions makes no warranties, either express or implied, as to any matter not expressly set forth in the license agreement, including without limitation the condition of the software, merchantability, or fitness for any particular purpose.

ITT Visual Information Solutions shall not be liable for any direct, consequential, or other damages suffered by the Licensee or any others resulting from use of the software packages or their documentation.

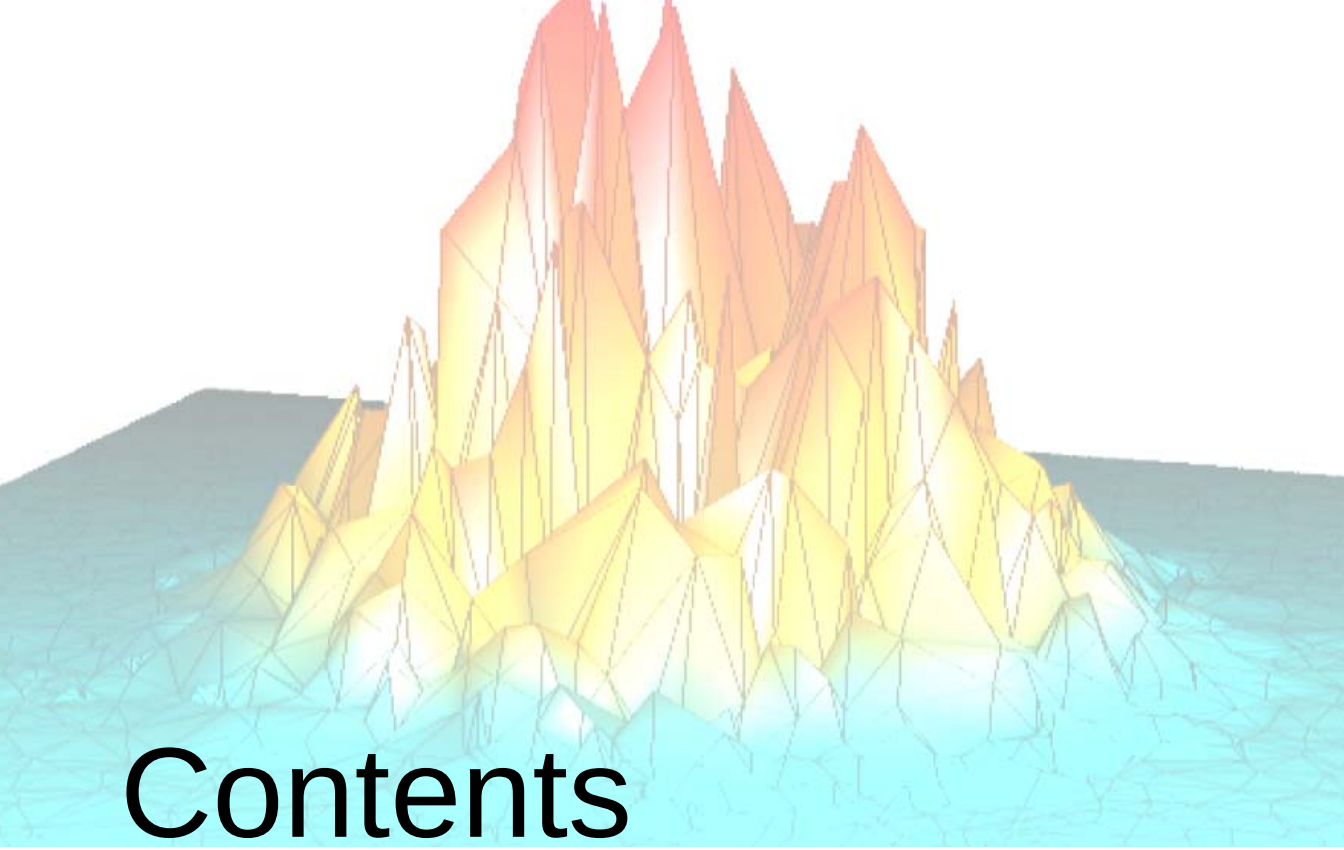
Permission to Reproduce this Manual

If you are a licensed user of these products, ITT Visual Information Solutions grants you a limited, nontransferable license to reproduce this particular document provided such copies are for your use only and are not sold or distributed to third parties. All such copies must contain the title page and this notice page in their entirety.

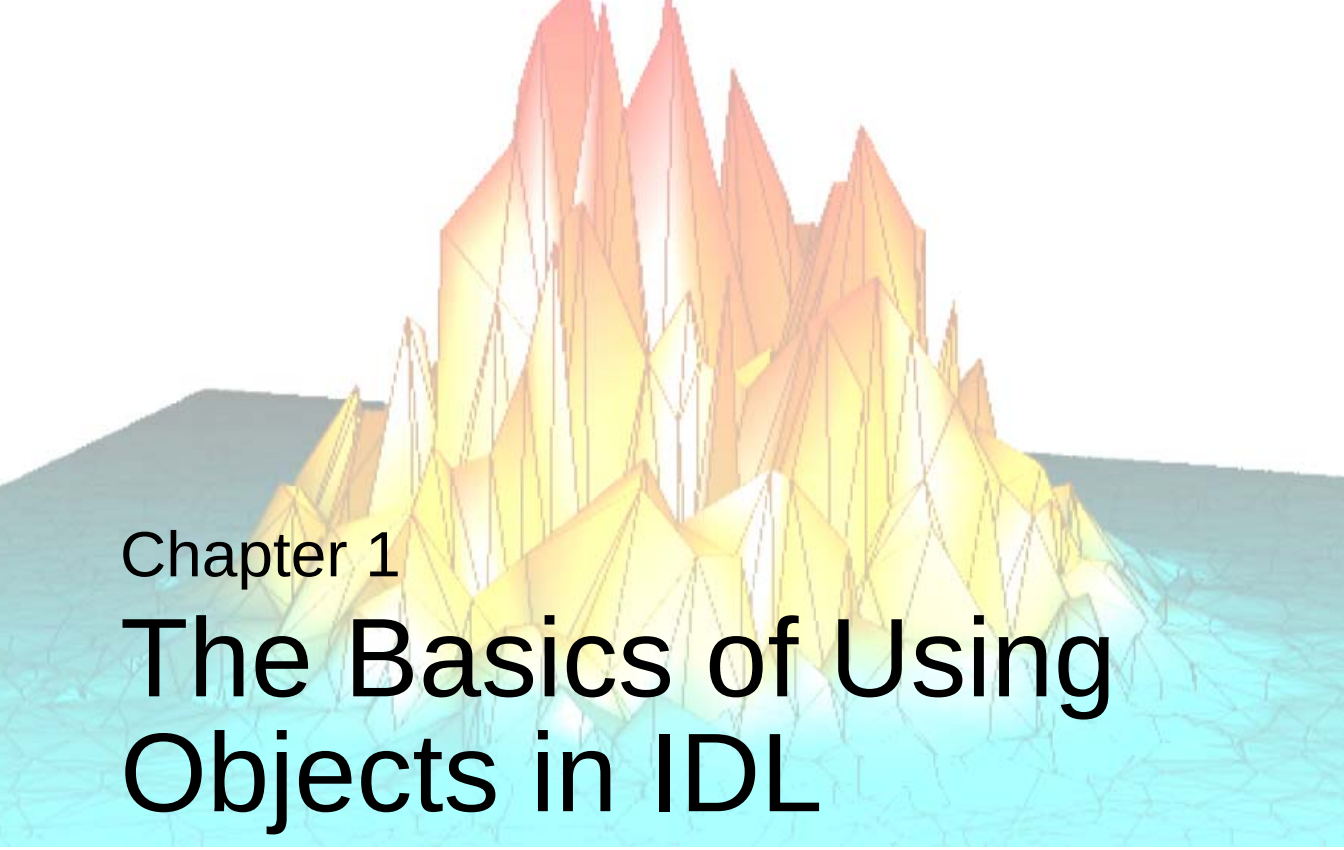
Export Control Information

This software and its associated documentation are subject to the controls of the Export Administration Regulations (EAR). It has been determined that this software is classified as EAR99 under U.S. Export Control laws and regulations, and may not be re-transferred to any destination expressly prohibited by U.S. laws and regulations. The recipient is responsible for ensuring compliance to all applicable U.S. Export Control laws and regulations.

Acknowledgments



Contents



Chapter 1

The Basics of Using Objects in IDL

The following topics are covered in this chapter:

Object-Oriented Programming Concepts

Traditional programming techniques make a strong distinction between routines written in the programming language (procedures and functions in the case of IDL) and data to be acted upon by the routines. *Object-oriented* programming begins to remove this distinction by melding the two into *objects* that can contain both routines and data. Object orientation provides a layer of abstraction that allows the programmer to build robust applications from groups of reusable elements.

Beginning in version 5.0, IDL provides a set of tools for developing object-oriented applications. IDL's Object Graphics engine is object-oriented, and a class library of graphics objects allows you to create applications that provide equivalent graphics functionality regardless of your (or your users') computer platform, output devices, etc. As an IDL programmer, you can use IDL's traditional procedures and functions as well as the new object features to create your own object modules. Applications built from object modules are, in general, easier to maintain and extend than their traditional counterparts.

This chapter describes how to create, configure and destroy inherent IDL graphic objects. For information on how to create and use custom object that you create, see Chapter 13, "Creating Custom Objects in IDL". If you are developing a custom iTool or components of an iTool (such as an operation or manipulator) see the *iTool Programming* for complete details and examples.

A complete discussion of object orientation is beyond the scope of this book—if you are new to object oriented programming, consult one of the many references on object oriented program that are available.

Using IDL Objects

The IDL Object Graphics system is a collection of pre-defined object classes, each of which is designed to encapsulate a particular visual representation. Actions (such as the modification of attributes, or data picking) may be performed on instances of these object classes by calling corresponding pre-defined methods. These objects are designed for building complex three-dimensional data visualizations.

For example, the IDLgrAxis object provides an encapsulation of all of the components associated with a graphical representation of an axis. One of the actions that can be performed on an axis is retrieving the current value of one or more of its attributes (such as its color, tick values, or data range). This action may be performed via the IDLgrAxis::GetProperty method. See "Graphic Objects—Visualization" in the functional category "Object Class Library" (*IDL Quick Reference*) for a complete listing of these types of objects.

Object Graphics should be thought of as a collection of building blocks. In order to display something on the screen, the user selects the appropriate set of blocks and puts them together so that as a group they provide a visual result. In this respect, Object Graphics are quite different than Direct Graphics. A single line of code is unlikely to produce a complete visualization. Furthermore, a basic understanding of the IDL object system is required (for instance, how to create an object, how to call a method, how to destroy an object, etc.). Because of the level at which these objects are presented, Object Graphics are aimed at application programmers rather than command line users.

Object Graphics do not interact in any way with the system variables (such as !P, !X, !Y, and !Z). Each graphic object is intended to encapsulate all of the information required to fully describe itself. Reliance on external structures is not condoned. The advantage of this approach is that once an object is created, it will always behave in the same way even if the system state is modified by another program, or if the object is moved to another user's IDL session, where the system state may have been customized in a different way than the state in which the object was originally defined.

Object Graphics are designed for building interactive three-dimensional visualization applications. Direct manipulation tools (such as the Trackball object) are provided to aid the application developer. Selection and data picking are also built in, so the developer can spend less time working out data projection issues and more time focusing on domain specific data analysis and visualization features. The IDL Intelligent Tools (iTools) are good examples of currently available applications built using Object Graphics. For more information, see the *iTools User's Guide*. Additional examples based on Object Graphics can be found in the IDL demo.

Creating Objects

To create an object from the IDL object class library, use the OBJ_NEW function. See "OBJ_NEW" (*IDL Reference Guide*). The Init method for each class describes the arguments and keywords available when you are creating a new object.

For example, to create a new object from the IDLgrAxis class, use the following call to OBJ_NEW along with the arguments and keywords accepted by the IDLgrAxis::Init method:

```
myAxis = OBJ_NEW('IDLgrAxis', DIRECTION = 1, RANGE = [0.0, 40.0])
```

When you create an object, it is *persistent*, meaning it exists in memory until you destroy it. You use an object reference (`myAxis`) to access the data associated with the object. This object reference actually accesses an object heap variable. (See "Object Heap Variables" on page 304 for details.)

Once an object has been created, you can access and modify it as needed. (See "The Object Lifecycle" on page 307 for additional information.) However, you should always explicitly clean up object references before ending a program. See "Destroying Objects" on page 26 for more information.

Acting on Objects Using Methods

In order to perform an action on an object's instance data, you must call one of the object's *methods*. In addition to their own specific methods, all object classes shipped with IDL except for the IDL_Container class have four methods in common: Cleanup, Init, GetProperty, and SetProperty. The Cleanup and Init methods are life-cycle methods, and cannot be called directly except within a subclass' Cleanup or Init method. (See "The Object Lifecycle" on page 307.) The GetProperty and SetProperty methods allow you to inspect (get) or change (set) the various properties associated with a given object. See "Modifying Object Properties" on page 22 for details.

To call a method, you must use the method invocation operator,-> (the hyphen followed by the greater-than sign).

Object Method Syntax

In the *IDL Reference Guide*, the Syntax section of each object method shows the proper syntax for calling the method.

Procedure Methods

IDL procedure methods have the syntax:

Obj->Procedure_Name, *Argument* [, *Optional_Arguments*]

where *Obj* is a valid object reference, *Procedure_Name* is the name of the procedure method, *Argument* is a required parameter, and *Optional_Argument* is an optional parameter to the procedure method. The square brackets around optional arguments are not used in the actual call to the procedure, they are simply used to denote the optional nature of the arguments within this document.

Function Methods

IDL function methods have the syntax:

Result = *Obj->Function_Name*(*Argument* [, *Optional_Arguments*])

where *Obj* is a valid object reference, *Result* is the returned value of the function method, *Function_Name* is the name of the function method, *Argument* is a required parameter, and *Optional_Argument* is an optional parameter. The square brackets around optional arguments are not used in the actual call to the function, they are simply used to denote the optional nature of the arguments within this document.

Note

All arguments and keywords to functions should be supplied *within* the parentheses that follow the function's name.

Arguments

The *Arguments* section describes each valid argument to the method.

Note

These arguments are positional parameters that must be supplied in the order indicated by the method's syntax.

Named Variables

Often, arguments that contain values upon return from the function or procedure method ("output arguments") are described as accepting "named variables." A named variable is simply a valid IDL variable name. This variable *does not* need to be defined before being used as an output argument. Note, however that when an argument calls for a named variable, only a named variable can be used—sending an expression causes an error.

Keywords

The *Keywords* section describes each valid keyword argument to the method.

Note

Keyword arguments are formal parameters that can be supplied in any order.

Keyword arguments are supplied to IDL methods by including the keyword name followed by an equal sign ("=") and the value to which the keyword should be set. Note that keywords can be abbreviated to their shortest unique length. For example, the XSTYLE keyword can be abbreviated to XST.

Note

In the case of Init, GetProperty and SetProperty methods, keywords often correspond to object *properties*. See "Modifying Object Properties" on page 22 for additional discussion.

Setting Keywords

When the documentation for a keyword says something similar to, “Set this keyword to enable logarithmic plotting,” the keyword is simply a switch that turns an option on and off. In general, setting such keywords equal to 1 (or using the /KEYWORD syntax) causes the option to be turned on. Explicitly setting the keyword to zero (or not including the keyword) turns the option off.

Modifying Object Properties

Some IDL objects have *properties* associated with them — things like color, line style, size, and so on. Properties are set or changed by supplying property-value pairs in a call to the object class' Init or SetProperty method:

```
Obj->OBJ_NEW('ObjectClass', PROPERTY = value, ... )
```

or

```
Obj->SetProperty, PROPERTY = value, ...
```

where *PROPERTY* is the name of a property and *value* is the associated property value.

Property values are retrieved by supplying property-value pairs in a call to the object class' GetProperty method:

```
Obj->GetProperty, PROPERTY = variable, ...
```

where *PROPERTY* is the name of a property and *variable* is the name of an IDL variable that will hold the associated property value.

Note

Property-value pairs behave in exactly the same way as Keyword-value pairs. This means that you can set the value of a boolean property to 1 by preceding the name of the property with a "/" character. The following are equivalent:

```
Obj->SetProperty, PROPERTY = 1

Obj->SetProperty, /PROPERTY
```

If you are familiar with IDL Direct Graphics, you will note that many of the properties of IDL objects correspond to keywords to the Direct Graphics routines. Unlike IDL Direct Graphics, the IDL Object Graphics system allows you to change the value of an object's properties without re-creating the entire object. Objects must be redrawn, however, with a call to the destination object's Draw method, for the changes to become visible.

Properties and the Property Sheet Interface

In addition to being able to set and change object property values programmatically, IDL provides a way for users to change property values via a graphical user interface. The WIDGET_PROPERTYSHEET function creates a user interface that allows users to select and change property values using the mouse and keyboard.

For an object property to be displayed in a property sheet, the property must be registered.

See "Registered Properties" (Chapter 28, *IDL Reference Guide*) for additional discussion.

Setting Properties at Initialization

Often, you will set an object's properties when creating the object for the first time, which is done by specifying any keywords to the object's Init method directly in the call of OBJ_NEW that creates the object. For example, suppose you are creating a plot and wish to use a red line to draw the plot line. You could specify the COLOR keyword to the IDLgrPlot::Init method directly in the call to OBJ_NEW:

```
myPlot = OBJ_NEW('IDLgrPlot', xdata, ydata, COLOR = [255, 0, 0])
```

In most cases, an object's Init method cannot be called directly. Arguments to OBJ_NEW are passed directly to the Init method when the object is created.

For some graphics objects, you can specify a keyword that has the same meaning as an argument. In Object Graphics, the value of the keyword overrides the value set by the argument. For example,

```
myPlot = OBJ_NEW('IDLgrPlot', xdata, ydata, DATAX = newXData)
```

The Plot object uses the data in `newXData` for the plot's X data.

Setting Properties of Existing Objects

After you have created an object, you can also set its properties using the object's SetProperty method. For example, the following two statements duplicate the single call to OBJ_NEW shown above:

```
myPlot = OBJ_NEW('IDLgrPlot', xdata, ydata)
myPlot->SetProperty, COLOR = [255, 0, 0]
```

Note

Not all keywords available when the object is being initialized are necessarily available via the SetProperty method. Keywords available when using an object's SetProperty method are noted with the word "Set" in the table included after the text description of the property.

Retrieving Property Settings

You can retrieve the value of a particular property using an object's GetProperty method. The GetProperty method accepts a list of keyword-variable pairs and returns the value of the specified properties in the variables specified. For example, to return the value of the COLOR property of the plot object in our example, use the statement:

```
myPlot->GetProperty, COLOR = plotcolor
```

The value of the COLOR property is returned in the IDL variable `plotcolor`.

You can retrieve the values of all of the properties associated with a graphics object by using the ALL keyword to the object's GetProperty method. The following statement:

```
myPlot->GetProperty, ALL = allprops
```

returns an anonymous structure in the variable `allprops`; the structure contains the values of all of the retrievable properties of the object.

Note

Not all keywords available when the object is being initialized are necessarily available via the GetProperty method. Keywords available when using an object's GetProperty method are noted with the word "Get" in the table included after the text description of the property.

About Object Property Descriptions

In the documentation for the IDL object class library, the description of each class is followed by a section describing the properties of the class. Each property description is followed by a table that looks like this:

Property Type	Boolean		
Name String	Hide		
Get: Yes	**Set:** No	**Init:** Yes	**Registered:** Yes

where

- **Property Type** describes the property type associated with the property. If the property is *registered*, the property type will be one of a number of registered property data types. If the property is not registered, this field will describe the generic IDL data type of the property value.

- **Name String** is the default value of the Name property attribute. If the property is registered, this is the value that appears in the left-hand column when the property is displayed in a property sheet widget. If the property is not registered, this field will contain the words *not displayed*.
- **Get**, **Set**, and **Init** describe whether the property can be specified as a keyword to the GetProperty, SetProperty, and Init methods, respectively.
- **Registered** describes whether the property is registered for display in a property sheet widget.

See Registered Property Data Types and "Registered Properties" (Chapter 28, *IDL Reference Guide*) for additional information.

Destroying Objects

Use the OBJ_DESTROY procedure to destroy an object. When an object is created using OBJ_NEW, memory is reserved for the object on the heap (see “Object Heap Variables” on page 304 for details). You must explicitly destroy objects in order to clean up the reference and the remove the data from memory. Objects are released as with a call to OBJ_DESTROY. Internally, this calls the object’s Cleanup method (see “Destruction” on page 309 for details).

For example, if you have created an axis object called myAxis, use the following syntax to clean up the object reference:

```
OBJ_DESTROY, myAxis
```

See “OBJ_DESTROY” (*IDL Reference Guide*) for further details.

Using Operations with Objects

Object reference variables are not directly usable by many of the operators, functions, or procedures provided by IDL. You cannot, for example, do arithmetic on them or plot them. You can, of course, do these things with the contents of the structures contained in the object heap variables referred to by object references, assuming that they contain non-object data.

There are four IDL operators that work with object reference variables: assignment, method invocation (described in "Acting on Objects Using Methods" on page 19), EQ, and NE. The remaining operators (addition, subtraction, etc.) do not make any sense for object references and are not defined.

Note

The structure dot operator (.) is allowed *within methods of a class* of a custom object. See "The Implicit Self Argument" on page 311 for details.

Many non-computational functions and procedures in IDL do work with object references. Examples are SIZE, N_ELEMENTS, HELP, and PRINT. It is worth noting that the only I/O allowed directly on object reference variables is default formatted output, in which they are printed as a symbolic description of the heap variable they refer to. This is merely a debugging aid for the IDL programmer—input/output of object reference variables does not make sense in general and is not allowed. Please note that this does *not* imply that I/O on the contents of non-object instance data contained in heap variables is not allowed. Passing non-object instance data contained in an object heap variable to the PRINT command is a simple example of this type of I/O.

You can also get information about an object as described in "Returning Object Type and Validity" on page 79.

Object Assignment

Assignment works in the expected manner—assigning an object reference to a variable gives you another variable with the same reference. Hence, after executing the statements:

```
;Define a class structure.
struct = { cname, data1:0.0 }

;Create an object.
A = OBJ_NEW('cname')
```

```
;Create a second object reference.
B = A

HELP, A, B
```

IDL prints:

```
A               OBJREF    = <ObjHeapVar1(CNAME)>
B               OBJREF    = <ObjHeapVar1(CNAME)>
```

Note that both A and B are references to the same object heap variable.

Object Equality and Inequality

The EQ and NE operators allow you to compare object references to see if they refer to the same object heap variable. For example:

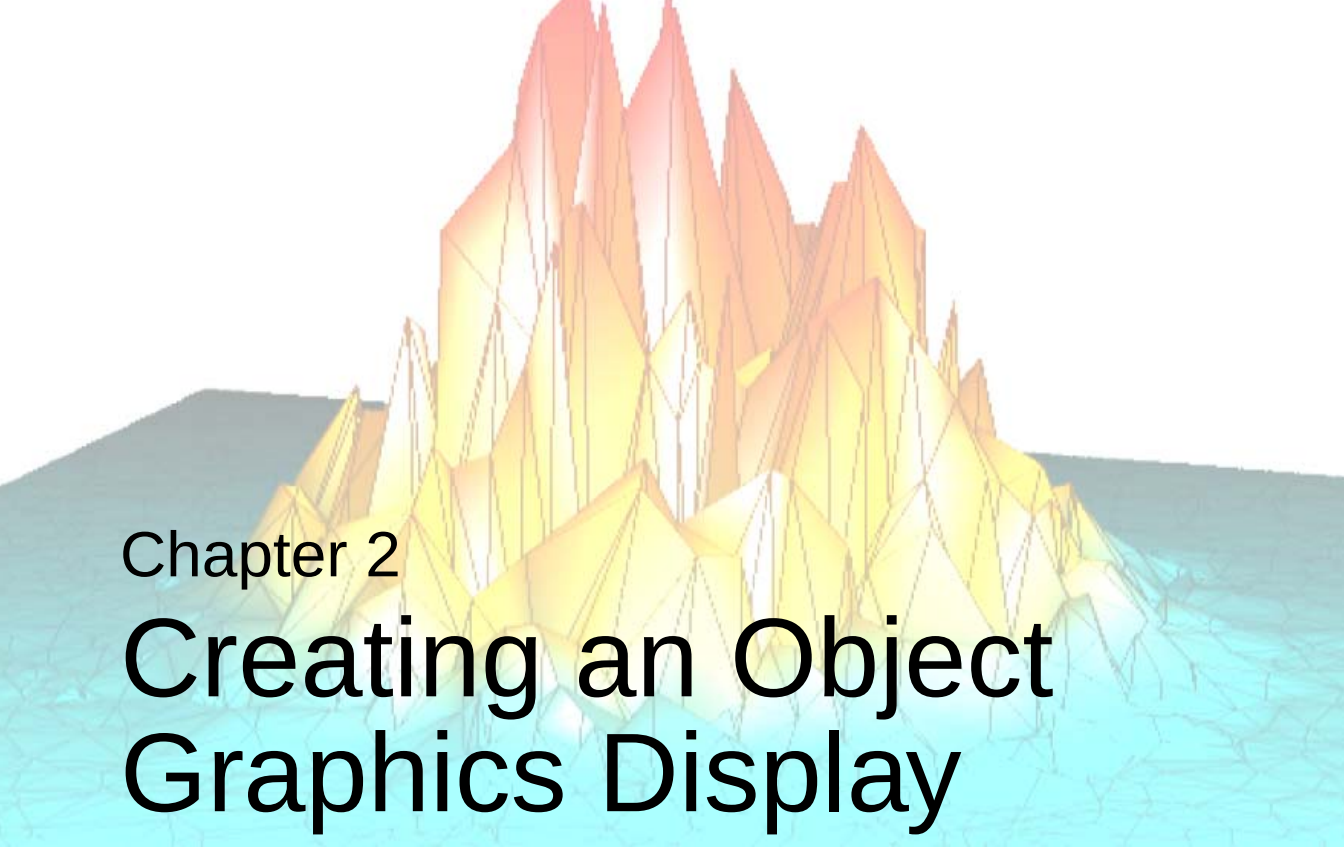
```
;Define a class structure.
struct = {cname, data:0.0}

;Create an object.
A = OBJ_NEW('CNAME')

;B refers to the same object as A.
B = A

;C contains a null object reference.
C = OBJ_NEW()

PRINT, 'A EQ B: ', A EQ B & $
PRINT, 'A NE B: ', A NE B & $
PRINT, 'A EQ C: ', A EQ C & $
PRINT, 'C EQ NULL: ', C EQ OBJ_NEW() & $
PRINT, 'C NE NULL:', C NE OBJ_NEW()
```

IDL prints:

```
A EQ B:     1
A NE B:     0
A EQ C:     0
C EQ NULL:  1
C NE NULL:  0
```

Object Examples

We have included a number of examples of object-oriented programming as part of the IDL distribution. Many of the examples used in this volume are included — sometimes in expanded form — in the `examples/doc/objects` subdirectory of the IDL distribution. By default, this directory is part of IDL's path; if you have not changed your path, you will be able to run the examples as described here. See "!PATH" (*IDL Reference Guide*) for information on IDL's path.



Chapter 2

Creating an Object Graphics Display

This chapter discusses creating and configuring Object Graphic displays.

Overview of Object Graphics Classes

The following sections provide an overview of the different types of objects included in the IDL Object Graphics class library. In order to describe the attributes of the IDL Object Graphics classes, we have grouped the objects into functional categories: Display Objects, Visualization Objects, Destination Objects, and File Format Objects.

Note

These category names are purely descriptive; for example, display objects contain the IDLgrModel, IDLgrScene, and IDLgrView classes, but no class named display.

See "Object Graphics Display Hierarchy" on page 35 for a discussion of the object tree, which shows the relationships between object classes.

There is some commonality among visualization object properties Following sections provide information about common properties including color, depth-buffering (how objects are layered in a view), and alpha-channel setting (transparency).

Naming Conventions

In general, object classes shipped with IDL have names of the form:

```
IDLxxYyyy
```

where *xx* represents the broad functional grouping (*gr* for graphics objects, *db* for database objects, and *an* for analysis, for example). *Yyyy* is the class name itself (such as *Axis* or *Surface*). Object classes that are useful in more than one functional context (container objects, for example) omit the functional grouping code entirely (IDL_Container). All object classes shipped with IDL are prepended with the letters IDL—we strongly suggest that you do not use this prefix when writing your own object classes, as we will continue to add new object classes using this convention.

The typographical convention used to describe IDL objects is slightly different from that used for non-object functions and procedures. Whereas non-object procedures are presented in upper case letters, object classes and methods use mixed case. For example, we refer to the PLOT routine, but to the IDLgrPlot object. Method names are also presented in mixed case (IDLgrAxis::GetProperty).

Creating an Object Graphics Display

All Object Graphics applications require at least two basic building blocks. These include:

- A destination object - the device (such as a window, memory buffer, file, clipboard, or printer) to which the visualization is to be rendered.
- A view object - the viewport rectangle (within the destination) within which the rendering is to appear (as well as how data should be projected into that rectangle).

For example:

```
; Create a destination object, in this case a window:
oWindow = OBJ_NEW('IDLgrWindow')
; Create a viewport that fills the entire window:
oView = OBJ_NEW('IDLgrView')
; Draw the view within the window:
OWindow->Draw, oView
```

By themselves, a window and a single view are not particularly enlightening, but you will find that these two types of objects are utilized by all Object Graphics applications. To change an attribute of an object, you do not have to create a new instance of that object. Instead, use the SetProperty method on the original object to modify the value of the attribute.

For example, to change the color of the view to gray:

```
; Set the color property of the view:
OView->SetProperty, COLOR=[60,60,60]
; Redraw:
OWindow->Draw, oView
```

If more than one view is to be drawn to the destination, then an additional object is required:

- A scene object - a container of views

For example:

```
; Create a scene and add our original view to it:
OScene = OBJ_NEW('IDLgrScene')
oScene->Add, oView
; Modify our original view so that it covers
; the upper left quadrant of the window.
OView->SetProperty, LOCATION=[0.0,0.5], DIMENSIONS=[0.5,0.5], $
   UNITS=3
; Create and add a second red view that covers
```

```
; the right half of the window.
OView2 = OBJ_NEW('IDLgrView', LOCATION=[0.5,0.0], $
   DIMENSIONS=[0.5,1.0], UNITS=3,COLOR=[255,0,0])
OScene->Add, oView2
; Now draw the scene, rather than the view, to the window:
OWindow->Draw, oScene
```

In the examples so far, the views have been empty canvases. For data visualization applications, these views will need some graphical content. To draw visual representations within the views, two additional types of objects are required:

- A model object - a transformation node
- A visualization graphic object - a graphical representation of data (such as an axis, plot line, or surface mesh). For more information, see "Visualization Objects" on page 40.

For example, to include a text label within a view:

```
; Create a model and add it to the original view:
oModel = OBJ_NEW('IDLgrModel')
oView->Add, oModel
; Create a text object and add it to the model:
oText = OBJ_NEW('IDLgrText','Hello World',ALIGNMENT=0.5)
oModel->Add, oText
; Redraw the scene:
OWindow->Draw, oScene
```

Notice that the scene, views, model, and text are all combined together into a self-contained hierarchy. It is the overall hierarchy that is drawn to the destination object.

The transformation associated with the model can be modified to impact the text it contains. For example:

```
; Rotate by 90 degrees about the Z-axis:
oModel->Rotate, [0,0,1], 90
; Redraw:
OWindow->Draw, oScene
```

When the objects are no longer required, they need to be destroyed. Destination objects must be destroyed separately, but the graphic hierarchies can be destroyed in full by simply destroying the root of the hierarchy. For example:

```
OBJ_DESTROY, oWindow
OBJ_DESTROY, oScene
```

In this example, the destruction of the scene will cause the destruction of all of its children (including the views, model, and text).

Object Graphics Display Hierarchy

An Object Graphics display can be thought of as a group of graphics objects organized into a hierarchy or tree. For example, a graphics object tree with four graphics atoms (visualization objects) might be contained in three separate model objects, which are in turn contained in two distinct view objects, both of which are contained in one scene object, which is the root of the graphics tree.

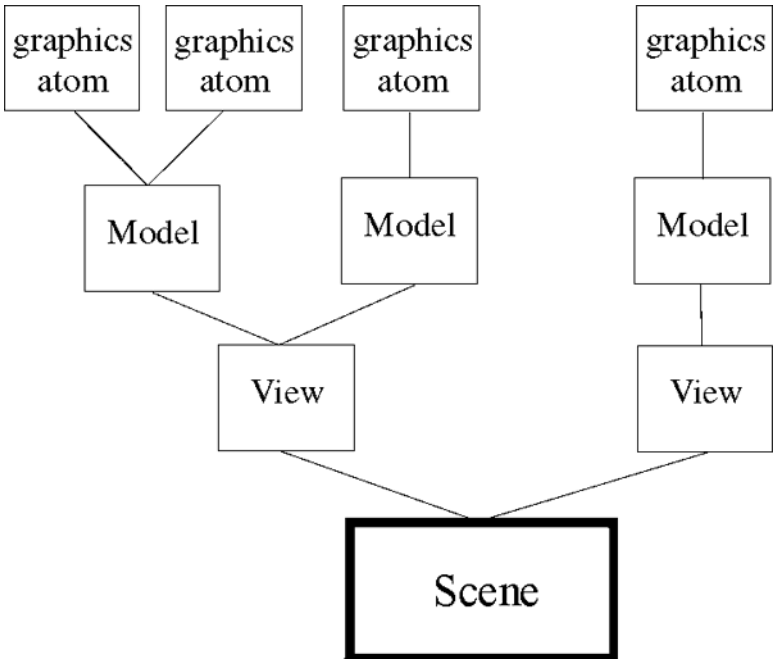


Figure 2-1: A Graphics Object Tree

Components of an Object Graphics Hierarchy

An object graphics display is commonly made up of the following components:

- **Destination objects** — a window, printer, clipboard or memory buffer that contains the display. One of these objects is required for any graphics tree. For more information, see “Destination Objects” on page 37. In the tree analogy, one of these objects is the ground.
- **Display objects** — a scene, view, or viewgroup that contains one or more models. Each model controls the spatial positioning of the visualization objects that it contains. See “Display Objects” on page 38.

  Note

  IDL_Container, like a view, can act as a container for other objects. Adding objects to a container object allows you to group disparate IDL objects into single object, and allows you to easily move or destroy the objects within the container. See “A Plotting Routine” on page 180 for an example that uses an IDL_Container object.

- **Visualization objects** — these low-level objects (shown as graphic atoms in Figure 2-1) are the used to create visualizations such as plot, contour, surface, and image displays. These objects contain data and have attributes such as size, color, or associated color palette. Visualization objects do not have an independent transformation matrix and do not contain other objects. See “Visualization Objects” on page 40 for more information.

Destination Objects

Destination objects are objects on which object trees can be rendered (displayed on a screen or printed on a printer). Detailed information about destination objects is available in Chapter 12, “Displaying, Copying and Printing Objects”.

Destination	Description
Buffer	Objects of the IDLgrBuffer class represent an off-screen, in-memory data area that may serve as a graphics source or destination.
Clipboard	Objects of the IDLgrClipboard class send Object Graphics to the operating system’s native clipboard or to a file in bitmap or vector format. See “Clipboard Objects” on page 275 for examples.
Printer	Objects of the IDLgrPrinter class represent a hardcopy graphics destination. By default, printer objects represent the default system printer; you can use the IDL routines DIALOG_PRINTJOB and DIALOG_PRINTERSETUP to change the printer associated with a printer object. See “Printer Objects” on page 277 for examples.
Window	Objects of the IDLgrWindow class represent an on-screen area on a display device in which graphic objects can be rendered. See “Window Objects” on page 267 for more information. Also see “Saving Window Contents to a File” on page 270 for information on how to save a view of displayed objects to an image file.

Table 2-1: Destination Objects

Note

When creating an iTool display, there is no need to manually configure a window object or destination objects. This is automatically done for you. See Chapter 3, “Visualizations” (*iTool User’s Guide*) for more information.

Display Objects

Minimally, you must have a view object in an Object Graphics display. However, it is likely that you will use a combination of the following display objects in any display. The “Object Graphics Display Hierarchy” on page 35 shows the relationship between these objects as a tree structure.

The advantage of organizing graphic objects into a tree structure is that by manipulating any of the branches of the tree, all of the sub-branches of that branch can be altered simultaneously. In Figure 2-1, changes to the spatial transformation associated with the model containing two graphics atoms will affect both of the visualization objects. Similarly, calling a window or printer object’s Draw method on the scene object will render all of the objects in the tree to that window or printer.

Object	Description
IDLgrScene	A scene, or instance of the IDLgrScene class, is the root-level object of most graphics trees. Instances of the IDLgrScene class have Add and Remove methods, which allow you to include or remove IDLgrView or IDLgrViewgroup objects in a scene. A scene object is one of the possible arguments for a destination object’s Draw method. It is not necessary to create a scene object if your graphics tree contains only one view object; in that case, the view can serve as the root of the tree.
IDLgrViewgroup	A viewgroup, or instance of the IDLgrViewgroup class, is a simple container object, similar to the Scene object. The Viewgroup differs from the Scene in two ways: 1. It will not cause an erase to occur on a destination when the destination object’s Draw method is called. 2. It can contain objects which do not have Draw methods. Viewgroups are designed to be placed within a scene, and therefor do not typically serve as the root-level object of a graphics tree. However, a viewgroup object can be an argument for a destination object’s Draw method. Instances of the IDLgrViewgroup class have Add and Remove methods, which allow you to include or remove objects in a viewgroup.

Table 2-2: Display Support Objects

Object	Description
IDLgrView	A view, or instance of the IDLgrView class, can serve as the root-level object of a graphics tree. Instances of the IDLgrView class have Add and Remove methods, which allow you to include or remove IDLgrModel objects in a view. A view object is one of the possible arguments for a destination object's Draw method. Every graphics tree must contain at least one view object. Often, it is convenient to divide the objects being rendered into separate views, which are then contained by a viewgroup or scene object.
IDLgrModel	A model, or instance of the IDLgrModel class, serves as containers for individual graphic objects (plot lines, axes, text, etc.) and for other model objects. Model objects include a three-dimensional transformation matrix that describes how the model and all of its components are positioned in space. Altering the model's transformation matrix changes the position and orientation of any objects the model contains. If a model object contains another model object, the contained model is positioned according to both its own transformation matrix and that of its container. See Chapter 3, "Positioning Objects in a View" for more information.

Table 2-2: Display Support Objects (Continued)

See "Creating an Object Graphics Display" on page 33 for an example that introduces the use of these objects. "Rendering Objects" on page 55 provides additional information.

"Mapping an Image onto Elevation Data" (Chapter 3, *Image Processing in IDL*) provides an example using the display objects to support texture-mapping.

Note

When creating an iTool display, there is no need to manually configure a window object or destination objects. This is automatically done for you. See Chapter 3, "Visualizations" (*iTool User's Guide*) for more information.

Visualization Objects

Visualization objects contain data that is designed to produce a visualization. These graphic objects are the basic drawable elements of the IDL Object Graphics system, and are container for other objects. Visualization objects are added to a model object, which controls the spatial positioning of all the objects it contains. Visualization objects combined in a model object (using the model object's Add method) share the same transformation matrix and can be rotated, scaled, or translated together.

Within the category of visualization objects, there is a sub-category of attribute objects. Attribute objects define the appearance of a visualization object, but themselves are not drawn, and thus do not need to be added to a model object. For example, an IDLgrFont object is associated with an IDLgrText object through the FONT property of the text object and defines the type characteristics of the text. Attribute objects are instances of one of the following classes: IDLgrFont, IDLgrPalette, IDLgrPattern, or IDLgrSymbol.

The following table introduces objects that are commonly see in different types of object graphics displays. Your display need not contain these specific combinations.

Display Type	Description
Plot	Objects of the IDLgrPlot class are individual plot lines, created from a user-supplied vector of dependent data values (and, optionally, a vector of independent data values). Plots do not automatically include axes. See Chapter 5, "Working with Plots and Graphs" for information on plot, symbol and axis objects. A plot display may include the following objects: • **Axis** — IDLgrAxis objects show data ranges (one object required for each axis to be rendered) • **Legend** — IDLgrLegend objects annotate individual data items or lines in a visualization. See "Legend Objects" on page 228. • **Colorbar** — IDLgrColorbar objects annotate the data values associated with colors used in a visualization. See "Colorbar Objects" on page 231. • **Symbol** — IDLgrSymbol objects define a graphical element that can be used when plotting data.
Contour	Objects of the IDLgrContour class are lines representing contour information plotted from user data. Contour displays, like plot display, may also include legend, colorbar, or symbol objects. See Chapter 5, "Working with Plots and Graphs". You can also use the following: • **Pattern** — IDLgrPattern objects defines which pixels are filled and which are left blank when a graphic object is filled. Patterns can be applied to successive contour levels.

Table 2-3: Visualization Object Displays

Display Type	Description
Image	Objects of the IDLgrImage class are two-dimensional arrays of data with an associated mapping of the data values to pixel values. See Chapter 4, “Working with Image Objects”. Displays containing image objects my also include: • **Palette** — IDLgrPalette objects define a color lookup table that maps indices to red, green, and blue values. • **ROI** — IDLgrROI objects are representations of a *region of interest*. Regions of interest are described as a set of vertices that may be connected to generate a path or a polygon, or may be treated as separate points. Objects of the IDLgrROIGroup class are representations of a group of regions of interest.
Surface	Objects of the IDLgrSurface class are individual three-dimensional surfaces, created from a user-supplied array of data values. See Chapter 6, “Working with Surface Objects”. • **Light** — IDLgrLight objects are light sources that illuminate visualization objects. Light objects are not actually rendered, but must be contained in a model object so that they can be positioned and transformed along with the graphic objects they illuminate. If no light object is included in a particular view, default lighting is supplied.
Volume	Objects of the IDLgrVolume class map a three-dimensional array of data values to a three-dimensional array of voxel colors, which, when drawn, are projected to two dimensions. Volume displays, like surface displays, can also include lights. See Chapter 7, “Creating Volume Objects”.

Table 2-3: Visualization Object Displays (Continued)

Display Type	Description
Polygon and Polyline	Polygons and polylines are low-level graphic objects that can be displayed by themselves or with other objects. See Chapter 8, "Polygon and Polyline Objects" for more information. Objects of the IDLgrPolygon class are individual polygons, created from a user-supplied array of data values. • **Tessellator** — IDLgrTessellator objects convert a simple concave polygon (or a simple polygon with holes) into a number of simple convex polygons (general triangles). Tessellation is useful because IDL's polygon object accepts only convex polygons. Objects of the IDLgrPolyline class are individual polylines, created from a user-supplied array of data points. Locations of the data points supplied are connected by a single line.
Text	Objects of the IDLgrText class are text strings that can be positioned within the rendering area. See "Text Objects" on page 219. • **Font** — IDLgrFont objects define the typeface, size, weight, and style of a text object with which it is associated. See "Font Objects" on page 223. Text objects are applicable to any of the previous displays.

Table 2-3: Visualization Object Displays (Continued)

See the "Graphic Objects—Visualization" category of the "Object Class Library" (*IDL Quick Reference*) for an alphabetical list of visualization objects.

Note

Objects of the TrackBall class provide a simple interface to allow the user to translate and rotate three-dimensional Object Graphics hierarchies displayed in an IDL WIDGET_DRAW window using the mouse. The trackball object translates widget events from a draw widget (created with the WIDGET_DRAW function) into transformations that emulate a virtual trackball (for transforming object graphics in three dimensions). See "Interactive 3D Transformations" on page 95 and "TrackBall" (*IDL Reference Guide*) for further details.

File Format Objects

File format object classes provide access to data stored within files of certain types. For example, the IDLffXMLSAX and IDLffXMLDOM classes provide access to attribute information stored in `.xml` files. The IDLffLangCat class also provides access to XML data. However, this object allows you to access XML data stored in language catalog files (`.cat`), which can be used to support internationalization. File format objects may or may not have a graphical element that can be displayed.

Format	Object Class Information
DICOM	Objects of the IDLffDICOM class contain the data for one or more images embedded in a DICOM Part 10 file. Use this object for read-only access to the data. The IDLffDicomEx class represents an extended IDL interface to DICOM format files, which includes read and write capabilities. The IDLffDicomEx object is available as a separately-purchased IDL module, and is described in *Medical Imaging in IDL*.
DXF	Objects of the IDLffDXF class contain geometry, connectivity and attributes for graphics primitives. **Note -** Also see "XDXF" (*IDL Reference Guide*) for information on directly displaying a `.dxf` file.
JPEG 2000	Objects of the IDLffJPEG2000 class provide an interface to files in the JPEG 2000 format.
Language Catalogs	Objects of the IDLffLangCat class provide an interface to IDL language catalog files. See Chapter 19, "Using Language Catalogs" (*Application Programming*) for usage details and examples.
Motion JPEG2000	Objects of the IDLffMJPEG2000 class provide a way to create and display Motion JPEG2000 animations. See Chapter 6, "Animations" (*Using IDL*) for more information.
MrSID	Objects of the IDLffMrSID class are used to query information about and load image data from a MrSID (`.sid`) image file.

Table 2-4: File Format Objects

Format	Object Class Information
MPEG	Objects of the IDLgrMPEG class allow you to save an array of image frames as an MPEG movie.
ShapeFiles	Objects of the IDLffShape class contain geometry, connectivity and attributes for graphics primitives accessed from ESRI Shapefiles.
VRML	Objects of the IDLgrVRML class allow you to save the contents of an Object Graphics hierarchy as a VRML 2.0 format file.
XML	**XML Parser** — Objects of the IDLffXMLSAX class represent an XML SAX level 2 parser. The XML parser allows you to read an XML file and store arbitrary data from the file in IDL variables. See Chapter 20, "Using the XML Parser Object Class" (*Application Programming*) for further details. **XML DOM** — The Document Object Model (DOM) describes the content of XML data in the form of a document object, which contains other objects that describe the various data elements of the XML document. Objects of the IDLffXMLDOM Classes classes represent items in an XML document; the items can be modified and the XML document file itself written to disk using these classes. See Chapter 21, "Using the XML DOM Object Classes" (*Application Programming*) for further details.

Table 2-4: File Format Objects (Continued)

See the "File Format Objects" category of the "Object Class Library" (*IDL Quick Reference*) for a list of file format objects.

Color in Object Graphics

Color in an Object Graphics display is the result of interaction between the color model defined for the destination object (e.g. window or printer), the destination object's inherent color model, and the color assigned to any visualization objects (e.g. plot, text or image objects) being displayed. This section explains how to specify color when using Object Graphics and how IDL interacts with the destination devices on which graphics are finally displayed.

Note

For general information on color systems (RGB, HSV, HLS, and CMY), and display color schemes (Indexed and RGB) see "Color Systems" or "Display Device Color Schemes" (Chapter 5, *Using IDL*).

Object Graphics supports two color models for newly created destination objects (such as an IDLgrWindow): an Indexed Color Model and an RGB Color Model. Indexed color allows you to map data values to color values using a color palette. RGB color allows you to specify color values explicitly, using an RGB triple. See "Indexed Color Model" on page 53 and "RGB Color Model in Object Graphics" on page 49.

Note

For some X11 display situations, IDL may not be able to support a color index model destination object in object graphics. We do, however, guarantee that an RGB color model destination will be available for all display situations.

The devices on which graphics are rendered—computer displays, printers, plotters, frame buffers, etc.—also support one or more color models. IDL performs any conversions necessary to support either the Indexed or RGB color model on any physical device. That is, the color model used by IDL is entirely independent of the color model used by the physical device. "How IDL Interprets Color Values" on page 53 explains how IDL's Object System color models interact with different device color models.

Note

You can specify the color of any graphic object using either a color index or red, green, and blue (RGB) value, regardless of the color model used by the destination object or the physical destination device. See "Specifying Object Color" on page 51 for details.

The majority of graphic visualization objects have a COLOR property that can be set to an indexed value or an RGB triple. You can set the color of any visualization object when it is first created and later change it using this property. In addition to the COLOR property, you can also associate a palette object (an instance of the IDLgrPalette class) with many visualization objects using the PALETTE property.

One exception is the IDLgrImage object, which does not have a COLOR property. Instead, you use the PALETTE property to specify a related color table for an indexed image, or set the INTERLEAVE property to define the arrangement of the image channels in a RGB image. Palette objects can also be associated with destination objects. See "Palette Objects" on page 50 for more information.

Color and Destination Objects

Each destination object has one of the two color models associated with it (an Indexed Color Model, and the RGB Color Model), shown in the following table.Once a destination object has been created, you cannot change the associated color model. You can, however, create destination objects that use different color models in the same IDL session. That is, it is possible to have two window objects—one using the Indexed color model and one using the RGB color model—on your computer screen at the same time.

Color Model	Keyword Value
INDEXED	COLOR_MODEL=1 See “Indexed Color Model in Object Graphics” on page 48.
RGB	COLOR_MODEL=0 (default) See “RGB Color Model in Object Graphics” on page 49

Table 2-5: Destination Object Color Models

You can specify the color of any graphic object using either a color index or an RGB value, regardless of the color model used by the destination object or the physical destination device. The main distinction between the two color models lies in how IDL manages the color lookup table (if any) of the physical destination device. See “How IDL Interprets Color Values” on page 53 for details.

A Note about Draw Widgets

Drawable areas created with the WIDGET_DRAW function deserve a special mention. When a draw widget is created with the GRAPHICS_LEVEL keyword set equal to 2, the widget contains an instance of an IDLgrWindow object rather than an IDL Direct Graphics drawable window. By default, the window object uses the RGB color model; to use the indexed color model, set the COLOR_MODEL keyword to WIDGET_DRAW equal to 1 (one).

Indexed Color Model in Object Graphics

In the Indexed color model, you have control over how colors are loaded into a color lookup table. If the Indexed Color Model is used, a color value (or individual image pixel) is expected to be an index into the palette associated with the destination object. To load a particular color table, create a palette object, then set it as a property

of the destination object in which the graphics are to be drawn (using the PALETTE keyword in the SetProperty method of the destination object). If a palette is not explicitly provided for a given destination object, a gray scale ramp is loaded by default.

When the contents of your destination object are rendered on the physical device (that is, when you call the Draw method for the destination object), the RGB values from the palette are either:

- passed directly through to the physical device (if it uses RGB values), or
- loaded into the physical device's lookup table (if it uses Indexed values).

Specify that a destination object should use the Indexed color model by setting the COLOR_MODEL property of the object equal to 1 (one):

```
myWindow = OBJ_NEW('IDLgrWindow', COLOR_MODEL = 1)
```

Specify a palette object by setting the PALETTE property equal to a palette object:

```
myWindow->SetProperty, PALETTE=myPalette
```

When you assign a color index to a visualization object that is drawn on the destination device, the color index is used to look up an RGB value in the specified palette. When you assign an RGB value to an object that is drawn on the destination device, the nearest match within the destination object's palette is found and used to represent that color.

See "How IDL Interprets Color Values" on page 53 for information on how a color assignment to a visualization object is interpreted by a destination object using either an RGB or Indexed color mode.

RGB Color Model in Object Graphics

In the RGB color model, IDL takes responsibility for filling the color lookup table on the destination device (if necessary). When the contents of your destination object are rendered on the physical device (that is, when you call the Draw method for the destination object), the RGB values are either:

- passed directly through to the physical device (if it uses RGB values), or
- matched as nearly as possible with colors loaded in the physical device's lookup table (if it uses Indexed values).

Specify that a destination object should use the RGB color model by setting the COLOR_MODEL property of the object equal to 0 (zero). This is the default color model value for newly created destination objects.

```
myWindow = OBJ_NEW('IDLgrWindow', COLOR_MODEL = 0)
```

Palette Objects

Objects of the IDLgrPalette class are used to create color lookup tables. Color lookup tables assign individual numerical values to color values; this allows you to specify the color of a graphic object with a single number (a color index) rather than explicitly providing the red, green, and blue color values (an RGB triple). Palettes are most useful when you want data values to correspond to color values—that is, if you want a data value of 200, for example, to always correspond to a single color. This correspondence is one of the main uses of the Indexed Color Model.

Creating Palette Objects

Specify three vectors representing the red, green, and blue values for the palette when you call the IDLgrPalette::Init method. The values in the red, green, and blue vectors must be integers between zero and 255, and the length of each vector must not exceed 256 elements. For example, the following statements create a palette object that reverses a standard grayscale ramp palette:

```
rval = (gval = (bval = REVERSE(INDGEN(256))))
myPalette = OBJ_NEW('IDLgrPalette', rval, gval, bval)
```

Using Palette Objects

Palettes can be associated either with graphics destination objects (windows or printers) or with individual graphic visualization objects:

```
myWindow->SetProperty, PALETTE=myPalette
```

or

```
myImage->SetProperty, PALETTE=myPalette
```

Note

Palettes associated with graphic visualization objects are only used when the destination object uses an RGB color model; if the destination object uses an indexed color model, the destination object's palette is always used.

See "IDLgrPalette::Init" (*IDL Reference Guide*) for details on creating palette objects and complete examples.

Specifying Object Color

The color of most graphic objects are specified by the COLOR property of that object. (The IDLgrImage object has a PALETTE property, not a COLOR property. See "IDLgrPalette::Init" (*IDL Reference Guide*) for examples.) In IDL Object Graphics, colors used for drawing visualization objects (such as an IDLgrText object) are typically represented in one of two ways:

- Indexed - a color is an index into a palette
- RGB - a color is a three-element vector, [red, green, blue]. See "Color Systems" (Chapter 5, *Using IDL*) for complete details.

You can set the color of an object either when the object is created or afterwards. For example, the following statement creates a view object and sets its color value to the RGB triple [60, 60, 60] (a dark gray).

```
myView = OBJ_NEW('IDLgrView', COLOR = [60, 60, 60])
```

The following statement changes the color value of an existing axis object to the color value specified for entry 100 in the color palette associated with the axis object.

```
myAxis->SetProperty, COLOR=100
```

The interpretation of this color depends upon the color model associated with the destination object, described in "Color and Destination Objects" on page 48.

Note

Remember that color palettes associated with individual graphic visualization objects are only used when the destination object uses an RGB color model. If the destination object uses an Indexed color model, the destination object's palette is always used.

Example Specifying RGB Values

RGB values are specified with RGB triples. An RGB triple is a three-element vector of integer values, [*r*, *g*, *b*], generally ranging between 0 and 255. A value of zero is the darkest possible value for each of the three channels—thus an RGB triple of [0, 0, 0] represents black, [0, 255, 0] represents bright green, and [255, 255, 255] represents white.

For example, suppose we create a plot line with the following statements:

```
myWindow = OBJ_NEW('IDLgrWindow')
myView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0, 0, 10, 10])
myModel = OBJ_NEW('IDLgrModel')
```

```
myPlot = OBJ_NEW('IDLgrPlot', FINDGEN(10), THICK = 5)
myModel->Add, myPlot
myView->Add, myModel
myWindow->Draw, myView
```

Notice the following aspects of the above example:

1. The newly-created window (destination) object uses an RGB color mode (the default).
2. The default color of the view object—the background against which the plot line is drawn—is white ([255, 255, 255]).
3. The default color of the plot object (and all objects, for that matter) is black.

Try changing the colors with the following statements:

```
myPlot->SetProperty, COLOR = [150, 0, 150]
myView->SetProperty, COLOR = [75, 250, 75]
myWindow->Draw, myView
```

To destroy the window and remove the objects created from memory, use:

```
OBJ_DESTROY, [myWindow, myView]
```

How IDL Interprets Color Values

IDL determines colors to display differently based on whether the destination object uses an Indexed or RGB color model, and on whether the physical destination device supports an Indexed or RGB color model.

Indexed Color Model

If the destination object uses an Indexed color model, the color displayed is calculated from the value specified by the object's COLOR property as follows:

If a Color Index is Specified

- If the physical device uses an Indexed color model, the specified color index is used as an index into the physical device's lookup table. (Remember that the physical device's color lookup table is loaded via the PALETTE keyword to the destination object.)
- If the physical device uses an RGB color model, the specified color index is used as an index into the destination object's palette. The RGB triple stored at the index's location in the palette is used as the physical device's color value.

If an RGB Triple is Specified

- If the physical device uses an Indexed color model, the RGB triple is mapped to the index of the nearest match in the device's color lookup table.
- If the physical device uses an RGB color model, the RGB triple is passed directly to the device.

RGB Color Model

If the destination object uses an RGB color model, the color displayed is calculated from the value specified by the object's COLOR property as follows:

If a Color Index is Specified

If the graphic object for which the color is being determined has a palette associated with it, the RGB triple at that palette's color index is retrieved. Otherwise, the RGB triple at the specified index in the destination object's palette is retrieved.

- If the physical device uses an Indexed color model, the RGB triple retrieved is mapped to the index of the nearest match in the device's color lookup table.

- If the physical device uses an RGB color model, the RGB triple retrieved is passed directly to the device.

If an RGB Triple is Specified

- If the physical device uses an Indexed color model, the RGB triple is mapped to the index of the nearest match in the device's color lookup table.
- If the physical device uses an RGB color model, the RGB triple is passed directly to the device.

If the RGB color model is used, the palette associated with a destination object does not necessarily have a one-to-one mapping to the hardware color lookup table for the device. For instance, the destination object may have a grayscale ramp loaded as a palette, but the hardware color lookup table for the device may be loaded with an even sampling of colors from the RGB color cube. When a user requests that a graphical object be rendered in a particular color, that object will appear in the nearest approximation to that color that the device can supply.

Rendering Objects

In Object Graphics, rendering occurs when the Draw method of a destination object is called. A scene, viewgroup, or view is typically provided as the argument to this Draw method. This argument represents the root of a graphics hierarchy. When the destination's Draw method is called, the graphics hierarchy is traversed, starting at the root, then proceeding to children in the order in which they were added to their parent.

For example, suppose we have the following hierarchy:

```
oWindow = OBJ_NEW('IDLgrWindow')
oView = OBJ_NEW('IDLgrView')
oModel = OBJ_NEW('IDLgrModel')
oView->Add, oModel
oXAxis = OBJ_NEW('IDLgrAxis', 0)
oModel->Add, oXAxis
oYAxis = OBJ_NEW('IDLgrAxis', 1)
oModel->Add, oYAxis
```

To draw the view (and its contents) to the window, the Draw method of the window is called with the view as its argument:

```
oWindow->Draw, oView
```

The window's Draw method will perform any window-specific drawing setup, then ask the view to draw itself. The view will then perform view-specific drawing (for example, clearing a rectangular area to a color), then calls the Draw method for each of its children (in this case, there is only one child, a model). The model's Draw method will push its transformation matrix on a stack, then step through each of its children (in the order in which they were added) and ask them to draw themselves. In this example, oXAxis will be asked to draw itself first; then oYAxis will be asked to draw itself. Once each of the model's children is drawn, the transformation matrix associated with the model is popped off of the stack.

Thus, for each object in the hierarchy, drawing essentially consists of three steps:

- Perform setup drawing for this object.
- Step through list of contained children and ask them to draw themselves.
- Perform follow-up drawing actions before returning control to parent.

The order in which objects are added to the hierarchy will have an impact on when the objects are drawn. Drawing order can be changed by using the Move method of a scene, viewgroup, view, or model to change the position of a specific object within the hierarchy.

The first time a visualization object (such as an axis, plot line, or text) is drawn to a given destination, a device-specific encapsulation of its visual representation is created and stored as a cache. Subsequent draws of this visualization object to the same destination can then be drawn very efficiently. The cache is destroyed only when necessary (for example, when the data associated with the visualization object changes). Graphic attribute changes (such as color changes) typically do not cause cache destruction. To gain maximum benefit from the caches, modification of object graphic properties should be kept to bare minimum.

Note

See "Performance Tuning Object Graphics" on page 66 for other performance enhancing strategies.

Simple Plot Example

The following section shows the IDL code used to create a simple object tree. While you are free to enter the commands shown at the IDL command line, remember that the IDL Object Graphics API is designed as a programmer's interface, and is not as well suited for interactive, ad hoc work at the IDL command prompt as are IDL Direct Graphics.

The following IDL commands construct a simple plot of an array versus the integer indices of the array. Note that no axes, title, or other annotations are included; the commands draw only the plot line itself. (This example is purposefully simple; it is meant to illustrate the skeleton of a graphics tree, not to produce a useful plot.)

```
; Create a view 2 units high by 100 units wide
; with its origin at (0,-1):
view = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0,-1,100,2])
; Create a model:
model = OBJ_NEW('IDLgrModel')
; Create a plot line of a sine wave:
plot = OBJ_NEW('IDLgrPlot', SIN(FINDGEN(100)/10))
; Create a window into which the plot line will be drawn:
window = OBJ_NEW('IDLgrWindow')
; Add the plot line to the model object:
model->ADD, plot
; Add the model object to the view object:
view->ADD, model
; Render the contents of the view object in the window:
window->DRAW, view
```

To destroy the window and remove the objects created from memory, use the following commands:

```
; Destroy the window and the view.
; Destroying the view object destroys all
; of the objects contained in the view:
OBJ_DESTROY, [window, view]
```

Controlling the Depth of Objects in a View

In graphics rendering, the depth buffer is an array of depth values maintained by a graphics device, one value per pixel, to record the depth of primitives rendered at each pixel. It is usually used to prevent the drawing of objects located behind other objects that have already been drawn in order to generate a visually correct scene. In IDL, smaller depth values are closer to the viewer.

Depth buffer properties provide more control over how Object Graphics primitives are affected by the depth buffer. You can now control which primitives may be rejected from rendering by the depth buffer, how the primitives are rejected, and which primitives may update the depth buffer.

Control of the depth buffer is achieved through a test function or by completely disabling the buffer. The depth test function is a logical comparison function used by the graphics device to determine if a pixel should be drawn on the screen. This decision is based on the depth value currently stored in the depth buffer and the depth of the primitive at that pixel location.

The test function is applied to each pixel of an object. A pixel of the object is drawn if the object's depth at that pixel passes the test function set for that object. If the pixel passes the depth test, the depth buffer value for that pixel is also updated to the pixel's depth value.

The possible test functions are:

- INHERIT - use the test function set for the parent model or view.
- NEVER - never passes.
- LESS - passes if the depth of the object's pixel is less than the depth buffer's value.
- EQUAL - passes if the depth of the object's pixel is equal to the depth buffer's value.
- LESS OR EQUAL - passes if the depth of the object's pixel is less than or equal to the depth buffer's value.
- GREATER - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value.
- NOT EQUAL - passes if the depth of the object's pixel is not equal to the depth buffer's value.

- GREATER OR EQUAL - passes if the depth of the object's pixel is greater than or equal to the depth buffer's value.
- ALWAYS - always passes

The IDL default is LESS. Commonly used values are LESS and LESS OR EQUAL, which allow primitives closer to the viewer to be drawn.

Disabling the depth test function allows all primitives to be drawn on the screen without testing their depth against the values in the depth buffer. When the depth test is disabled, the graphics device effectively uses the painter's algorithm to update the screen. That is, the last item drawn at a location is the item that remains visible. The depth test function of ALWAYS produces the same result as disabling the depth test.

Moreover, you can disable updating the depth buffer. Disabling depth buffer writing prevents the updating of depth information as primitives are drawn to the frame buffer. Such primitives are unprotected in the sense that any other primitive drawn later at that location will draw over it as if it were not there.

Most visualization objects now have the following properties related to the depth buffer:

- DEPTH_TEST_DISABLE
- DEPTH_TEST_FUNCTION
- DEPTH_WRITE_DISABLE

For more details on these properties, see each object's property list in the *IDL Reference Guide*.

Controlling Object Transparency

IDL objects which support an alpha channel are:

- IDLgrAxis
- IDLgrContour
- IDLgrImage
- IDLgrPlot
- IDLgrPolygon
- IDLgrPolyline
- IDLgrROI
- IDLgrSurface
- IDLgrSymbol
- IDLgrText
- IDLgrVolume

Note

The transparency of an IDLgrImage object can be defined using a band of data defining the alpha values, and/or the ALPHA_CHANNEL property. Regardless of which way the image transparency is defined, you also need to set BLEND_FUNCTION property. See “Defining Transparency in Image Objects” on page 115 for details.

The alpha channel has many uses. One of the most important is drawing primitives semi-transparently, which can be used to enhance your object graphics scene. An example might be a text label drawn semi-transparently to let other graphical details “show through” the text label. This would allow you to use a larger text font size, rather than using a small font size to squeeze text between details in a scene. Another use for alpha channel might be to draw polygons and surfaces semi-transparently, allowing you to see “inside” certain objects and structures.

Some of the most important uses for semi-transparent rendering are discussed in the following sections.

Opacity and Transparency

Opacity describes the degree to which an object blocks the appearance of other objects. In IDL, the value used for the ALPHA_CHANNEL properties in IDLgr* objects is a measure of the object's opacity. A value of 1.0 indicates complete opacity. The object completely blocks the appearance of other objects. Conversely, an opacity value of 0.0 indicates that the object does not block the appearance of objects at all. Intermediate values indicate varying degrees of visibility for covered objects.

Transparency is essentially the opposite of opacity. Transparency indicates the degree to which an object does not block the appearance of other objects. Complete or full transparency is indicated by an opacity value of 0.0, while an object that is not transparent at all has an opacity value of 1.0.

By default all IDLgr* graphic objects use an ALPHA_CHANNEL value of 1.0, indicating full opacity (zero transparency), matching the rendering behavior before the addition of the ALPHA_CHANNEL property. To change the opacity of the object, simply change the this property to a value between 0.0 (zero opacity or full transparency) and 1.0.

Blending Mathematics

Blending is the drawing of semi-transparent objects on a screen already containing objects. During rendering, the color of the pixels belonging to the primitive being rendered are blended with the color of the pixels that are already on the screen, producing the desired blending effect. This process is accomplished on a pixel-by-pixel basis.

IDL uses this well-established blending equation:

```
newColor = oldColor * (1 - alpha) + primitiveColor * alpha
```

An example might suppose that you want to draw a red square in an area of the screen that is completely green. By default, the alpha value is 1.0, so the result is:

```
[255, 0, 0] = [0, 255, 0] * (1.0 - 1.0) + [255, 0, 0] * 1.0
```

The green color is removed completely and replaced by red, the expected result of conventional non-blended rendering.

If the alpha value is changed to equal 0.5, the result is:

```
[127, 127, 0] = [0, 255, 0] * (1.0-0.5) + [255, 0, 0] * 0.5
```

The resulting color is the half of the red of the polygon combined with half of the green of the background, a pale yellow.

If you draw another red square in the same place with the same alpha, the red square is blended with the now current contents of the screen:

```
[190, 63, 0] = [127, 127, 0] * (1.0-0.5) + [255, 0, 0] * 0.5
```

Note

Large levels of semi-transparent rendering may reduce rendering performance. This is because the graphics blending operation that is performed involves reading the destination pixel from the frame buffer, combining it with the new color value and then writing the result back to the frame buffer. This is more expensive than simply overwriting the frame buffer contents with the new color value. The degree to which your performance will be impacted depends heavily on the hardware and software components of your graphics system.

Rendering Order

The colors of the pixels on the screen are important when drawing a blended primitive. Similarly, the order in which the primitives are drawn is also very important when drawing scenes with blended primitives.

In computer graphics, depth sorting presents a similar challenge. Without depth sorting, a scene would have to be drawn from back to front to obtain a correct result. IDL handles depth sorting by providing a "depth buffer" (also known as a "Z-buffer") allowing you to draw the primitives in any order while allowing the primitives closer to the viewer to still appear to be on top.

There is no similar feature for alpha-blended primitives. Be sure to draw the blended primitives carefully so that all primitives behind a blended primitive are drawn before the blended primitive.

If your scene consists of many primitives that are not blended and a few text labels that are drawn with blending, it is a good idea to defer the drawing of the labels until after everything else is drawn. This will allow users to see through all labels and to see the objects beneath. If a non-blended primitive is drawn on top of and after a blended primitive, it will cover the blended primitive. If any primitive is drawn behind but after a blended primitive, the primitive drawn later will not appear where the blended primitive covers it, due to depth buffering. In other words, it is not possible to blend primitives unless all objects behind the blended primitive which are to be blended are already drawn.

Note

If you have a complex scene where many primitives are blended, it may be difficult to determine the proper ordering.

Inter- and Intra-primitive Rendering Orders

Inter-primitive rendering order deals with the ordering of primitive objects within an IDLgrModel. For primitives which do not intersect each other, it is straightforward to order these in a back-to-front viewing order, particularly if your scene is fixed so it cannot be rotated by the user. This is done by arranging your primitives along the Z direction so that the objects farthest away appear first in the IDLgrModels, which makes them draw first.

If primitives intersect, it may be necessary to divide the object so that the back parts of each primitive are drawn first, and then the front parts. This can be a very difficult issue.

Intra-primitive rendering order deals with the ordering of graphical items within an IDL graphics primitive. Some primitives, such as IDLgrSurface and IDLgrPolygon actually consist of a large number of individual polygons. They are not all drawn at once, and the order in which they are drawn is also important when drawing with blending.

You can control the order in which the individual polygons are drawn in an IDLgrPolygon object by ordering the vertices or specifying the order in the POLYGONS property. Polygons specified first in the POLYGONS list are drawn first.

Viewing and Rotation

If you draw a typical height field with IDLgrSurface and invoke blending, the object might look right from some viewing orientations.

For example, try the following:

```
XOBJVIEW, OBJ_NEW('idlgrsurface',$
BESELJ(shift(dist(40),20,20)/2,0) * 20, STYLE=2,$
   ALPHA_CHANNEL=0.5)
```

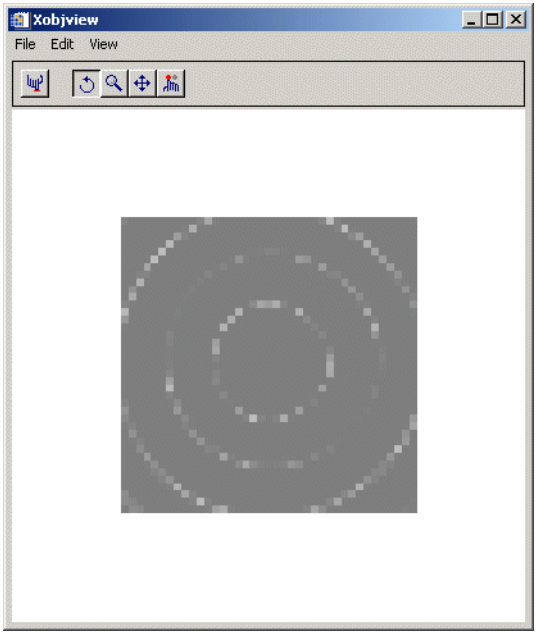


Figure 2-2: Viewing Alpha Channel in an Object

Notice in the previous figure that you can see-through the waves in the object to see other waves, but only when you view the object from certain directions. From other directions, all you see are the waves closer to you.

Solving this problem can be extremely difficult. A complete solution would generate a scene for every possible viewing angle, where the polygons are drawn back to front, splitting them if necessary. There are several techniques available for accomplishing this, one of them being the Binary Space Partition Tree, however this is not supported directly in IDL. If the objects are simple, it might be possible to construct a few scenes that give correct or passable results.

For example, if you wanted to look at a semi-transparent sphere from all angles, creating eight models might suffice. Each of the eight models contains the polygons sorted in back-to-front order for a viewing direction corresponding to each of the eight octants formed by the half spaces of the three principle axes. As the user rotates the scene with a trackball, the program would select the appropriate model, based on the current viewing direction. More complex scenes may require more models.

Depth Buffer Updating

For any value of the ALPHA_CHANNEL property, IDL updates the depth buffer when the primitive is drawn, unless the DEPTH_WRITE_DISABLE property is set to a value that disables depth buffer updates. Thus, even if you draw a completely transparent primitive, the depth buffer is updated as if there were a visible primitive drawn there. This means that subsequent primitives drawn behind the transparent object are not visible. Though potentially confusing, this can also be a useful way to hide objects in certain situations.

After drawing a transparent object, that there may be gaps in objects drawn later. For example, suppose lines in a plot are drawn with ALPHA_CHANNEL=0 (transparent), and then symbols are drawn. Where the symbols and lines intersect, there are gaps in the symbols. The gaps are caused by the invisible lines changing the depth buffer, thus masking out the symbols that are drawn later. At times, the ability to modify the depth buffer without changing the color buffer is a useful tool for clever clipping operations. In other contexts, you may consider using invisible polygons to mask out entire areas. However, if the partial or entire invisibility of objects drawn after a transparent object is unintended use one of the following options:

- Set the DEPTH_TEST_FUNCTION=4, or disable depth testing entirely using the DEPTH_WRITE_DISABLE property.
- Set the HIDE property to 1 if ALPHA_CHANNEL becomes 0.

Either of these options would erase the gaps in the symbols caused by the transparent plot lines as described in the previous situation.

Performance Tuning Object Graphics

The Object Graphics subsystem is designed to provide a rich set of graphical functionality that can be displayed in reasonable time. This section offers suggestions on how to adjust your system and programs to achieve the best rendering performance.

See the following topics for details:

- "Hardware vs. Software Rendering" on page 66
- "Polygon Optimization" on page 209
- "Optimizing Light Object Use" on page 235
- "Improving Window Drawing Performance" on page 272

Hardware vs. Software Rendering

The RENDERER property to the IDLgrWindow object (or associated platform-specific preferences in the IDL Preference system) allows you to select between the operating system's native (hardware) rendering system and a platform independent (software) rendering system for IDL Object Graphics displays.

Hardware rendering allows IDL to make use of 3D graphics accelerators that support OpenGL, if any are installed in the system. In general, such accelerators will provide better rendering performance for many object graphics displays. This is typically true for images rendered using texture-mapped polygons (the default behavior for IDLgrImage beginning with IDL 6.2).

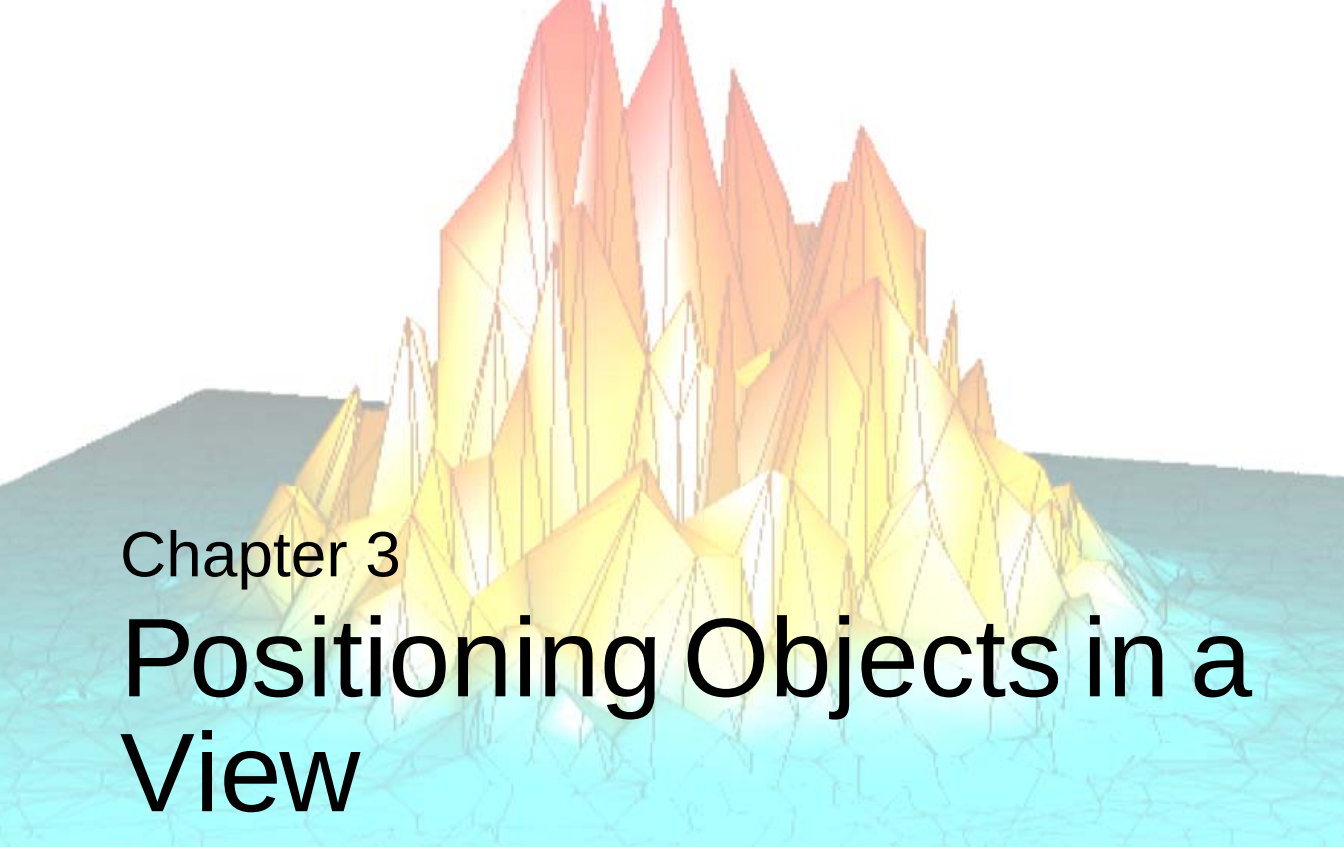
The software rendering system will generally run more slowly than the hardware rendering system. However, use of the software rendering system has a few important advantages:

- Software rendering is available in situations where hardware rendering is not (remote display to non-OpenGL capable X servers, for example).
- The number of expose events an IDL application will have to respond to is much smaller when software rendering is used.
- The software rendering system is generally much faster than the hardware rendering system for Instancing.
- Software rendering can be used to avoid bugs in third-party hardware rendering system driver software.

- Finally, on some displays (most notably SGI systems with 24 or fewer bitplanes), the quality of the screen display will be better when using the software rendering system because its design allows more bitplanes to be used.

Note

By default, IDL uses the renderer specified by the IDL_GR_WIN_RENDERER preference (Microsoft Windows) or the IDL_GR_X_RENDERER preference (UNIX). If your platform does not have a native OpenGL implementation, IDL uses its own software implementation regardless of the preference value or the value of the RENDERER property.



Chapter 3
Positioning Objects in a View

The following topics are covered in this chapter:

Positioning Visualizations in a View

Unlike IDL Direct Graphics, the IDL Object Graphics system does not automatically position and size the objects to be rendered. It is up to you, as a programmer, to properly define how your graphic elements will be positioned when rendered.

There are three aspects to this transformation from a generic depiction of your data to a representation that can be rendered to an output device (a graphics destination object, such as a window or printer) with the perspective, size, and location you want.

Viewport

The first aspect is the view of the graphics objects to be rendered: the size of the viewing area (the viewport), the type of projection used, the position of the viewer's eye as it looks at the graphics objects, and the particular view volume in three-dimensional space that will be rendered to the viewing area. These elements of the view of your graphics objects are, appropriately, controlled by properties of the IDLgrView object being rendered. See "Viewport" on page 71.

Location

The second aspect of the transformation is the location and position of your graphics objects with respect to the viewing area. Graphics objects can be translated, rotated, or scaled by setting the appropriate properties of the IDLgrModel object that contains them. See "Translating, Rotating and Scaling Objects" on page 91.

Note

The viewport and location of an object are independent: It is possible, for example, to translate a graphic object so that it is no longer within the viewing area that is rendered in a window or on a printer.

Coordinate Systems and Scaling

The third aspect of the transformation is the conversion between data, device, and normalized coordinates. The IDL Object Graphics system gives you full control over which data values are used, which are displayed, and which coordinate systems are used. This means that you must explicitly ensure that the objects to be rendered and the view object to which they belong use the same coordinate system and are scaled appropriately. This chapter discusses the properties and methods used to size and position both your viewing area and the graphics objects you wish to render. See "Converting Data to Normal Coordinates" on page 80.

Viewport

Several elements of an IDLgrView object control how objects appear when displayed:

- “Location and Dimension” on page 71 — define the viewport within the destination object
- “Projection” on page 73 — define either a parallel or perspective projection
- “Eye Position” on page 75 — define the distance of eye from the viewing plane for perspective projections
- “View Volume” on page 77 — define the view volume that is projected into the viewport

Location and Dimension

One of the first steps in determining how graphics objects will appear when rendered on a graphics destination object is to select the location and dimensions of the rectangular area—the viewport—on the destination in which the rendering will be displayed. Set the location and dimensions of the viewport using the LOCATION and DIMENSIONS keywords to the IDLgrView::Init method when creating the view object (or after creation using the SetProperty method).

For example, the following statement creates a view object with a viewport that is 300 pixels by 200 pixels, with its lower left corner located 100 pixels up from the bottom and 100 pixels to the right of the left edge of the destination object:

```
myView = OBJ_NEW('IDLgrView', LOCATION=[100,100], $
   DIMENSIONS=[300,200])
```

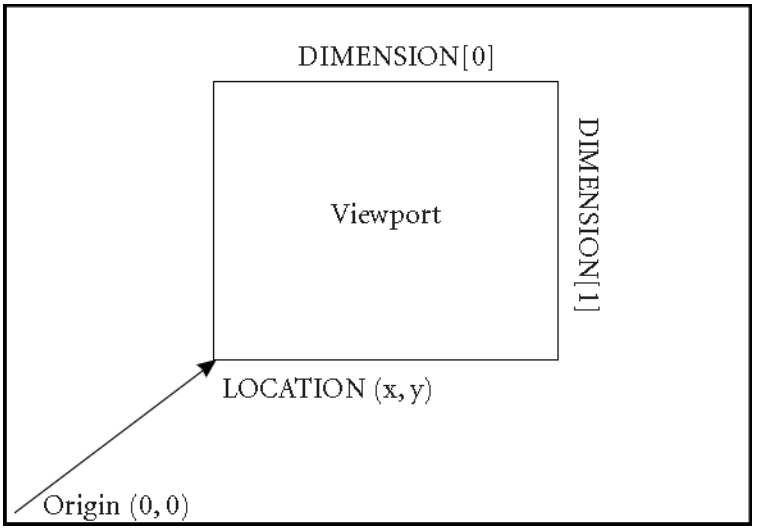


Figure 3-1: Positioning a View on the Screen

Both the LOCATION and DIMENSIONS properties of the view object honor the value of the UNITS property, which specifies the type of units in which measurements are made. (Pixels are the default units, so no specification of the UNITS keyword was necessary in the above example.)

The viewport of an existing view can be changed using the SetProperty method:

```
myView->SetProperty, LOCATION=[0,0], DIMENSIONS=[200,200]
```

changes the location of the viewport to have its lower left corner at (0, 0) and a size of 200 pixels by 200 pixels.

Note

The eye is positioned in only one dimension (along the *z*-axis) and always points in the –*z* direction.

Projection

When three-dimensional graphics are displayed on a flat computer screen or printed on paper, they must be projected onto the viewing plane. A projection is a way of converting positions in 3D space into locations in the 2D viewing plane. IDL supports two types of projections—parallel and perspective—for each view.

Parallel Projections

A parallel projection projects objects in 3D space onto the 2D viewing plane along parallel rays. The figure below shows a parallel projection; note that two objects that are the same size but at different locations still appear to be the same size when projected onto the viewplane.

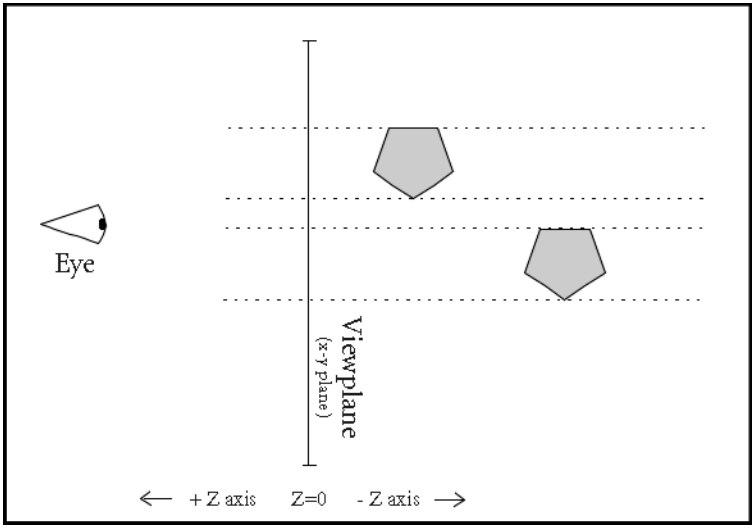


Figure 3-2: In a Parallel Projection, Rays Do Not Converge at the Eye

View objects use a parallel projection by default. To explicitly set a view object to use a parallel projection, set the PROJECTION keyword to the IDLgrView::Init method equal to 1 (or use the SetProperty method to set the projection for an exiting view object):

```
myView->SetProperty, PROJECTION = 1
```

Perspective Projections

A perspective projection projects objects in 3D space onto the 2D viewing plane along rays that converge at the eye position. The figure below shows a perspective projection; note that objects that are farther from the eye appear smaller when projected onto the viewplane.

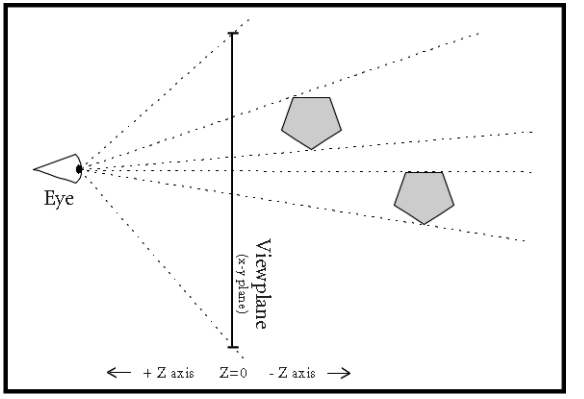


Figure 3-3: In a Perspective Projection, Rays Converge at the Eye

Set the PROJECTION keyword to the IDLgrView::Init method equal to 2 (or use the SetProperty method to set the projection for an exiting view object) to use a perspective projection:

```
myView->SetProperty, PROJECTION = 2
```

Eye Position

The eye position is the position along the *z*-axis from which a set of objects contained in a view object are seen. Use the EYE keyword to the IDLgrView::Init method to specify the distance from the eye position to the viewing plane (or use the SetProperty method to alter the eye position of an existing view object). The eye position must be a *z* value larger than the *z* value of the near clipping plane (see "Near and Far Clipping Planes" on page 77) or zero, whichever is greater. That is, the eye must always be located at a positive *z* value, and must be outside the volume bounded by the near and far clipping planes.

For example, the following moves the eye position to $z = 5$:

```
myView->SetProperty, EYE=5
```

The eye is always positioned directly in front of the center of the viewplane rectangle. That is, if the VIEWPLANE_RECT property is set equal to [–1, –1, 2, 2], the eye will be located at X=0, Y=0.

Changing the position of the eye has no effect when you are using a parallel projection. Changing the eye position when you are using a perspective projection has a somewhat counter-intuitive affect: moving the eye closer to the near clipping plane causes objects in the volume being rendered to appear smaller rather than larger. To understand why this should be true, consider the following diagram.

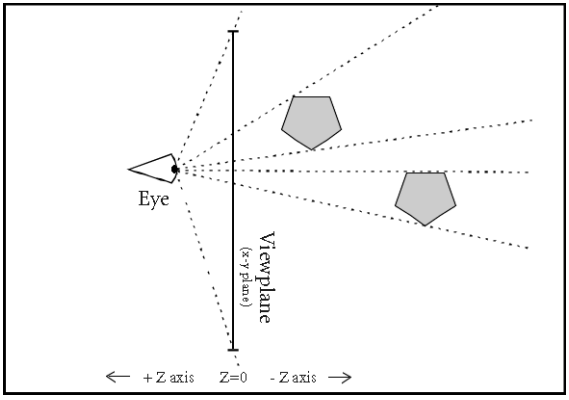


Figure 3-4: Moving the Eye Closer to the Viewplane Causes Objects to Appear Smaller

In a perspective projection, rays from the graphic objects in the view volume converge at the eye position. When the eye is close to the viewing plane, the projected rays cross the viewing plane (where rendering actually occurs) in a relatively small area. When the eye moves farther from the viewing plane, the projected rays become more nearly parallel and occupy a larger area on the viewing plane when rendered.

View Volume

The view volume defines the three-dimensional volume in space that, once projected, is to fit within the viewport. There are two parts to the view volume: the viewplane rectangle and the near and far clipping planes.

Viewplane Rectangle

The viewplane rectangle defines the bounds in the X and Y directions that will be mapped into the viewport. Objects (or portions of objects) that lie outside the viewplane rectangle will not be rendered. The viewplane rectangle is always located at Z=0.

Use the VIEWPLANE_RECT keyword to the IDLgrView::Init method (or use the SetProperty method if you have already created the view object) to set the location and extent of the viewplane rectangle. Set the keyword equal to a four-element floating-point vector; the first two elements specify the X and Y location of the lower left corner of the rectangle, and the second two elements specify the width and height. The default rectangle is located at (-1.0, -1.0) and is two units wide and two units high ([–1.0, –1.0, 2.0, 2.0]). For example, the following command changes the viewplane rectangle to be located at (0.0, 0.0) and to be one unit square:

```
myView->SetProperty, VIEWPLANE_RECT = [0.0, 0.0, 1.0, 1.0]
```

Note

See “Panning in Object Graphics” on page 111 for an example that modifies the VIEWPLANE_RECT to control what portion of an image is displayed in a view.

Near and Far Clipping Planes

The near and far clipping planes define the bounds in the Z direction that will be mapped into the viewport. Objects (or portions of objects) that lie nearer to the eye than the near clipping plane or farther from the eye than the far clipping plane will not be rendered. The figure below shows near and far clipping planes.

Use the ZCLIP keyword to the IDLgrView::Init method (or use the SetProperty method if you have already created the view object) to set the near and far clipping planes. Set the keyword equal to a two-element floating-point vector that defines the positions of the two clipping planes: [near, far]. The default clipping planes are at

Z = 1.0 and *Z* = –1.0 ([1.0, –1.0]). For example, the following command changes the near and far clipping planes to be located at *Z* = 2.0 and *Z* = –3.0, respectively.

```
myView->SetProperty, ZCLIP = [2.0, -3.0]
```

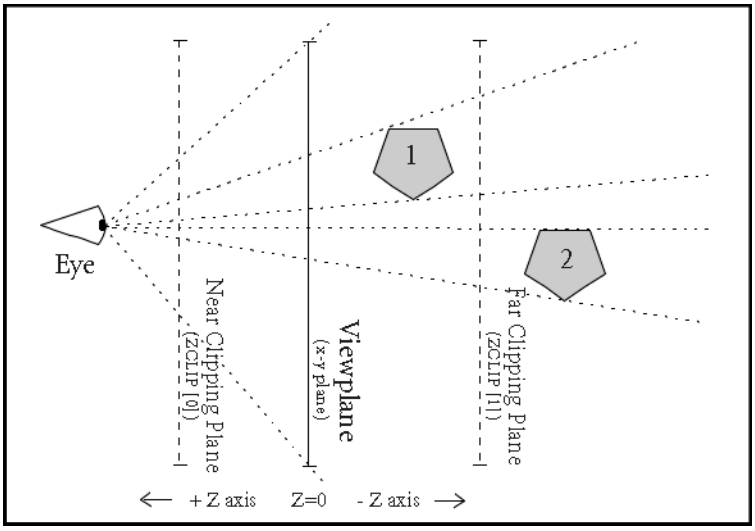


Figure 3-5: Near and Far Clipping Planes. Object 2 is not rendered, because it does not lie between the near and far clipping planes.

Finding an Appropriate View Volume

Finding an appropriate view volume for a given object tree is relatively simple in theory. To find the appropriate viewplane rectangle, you must find the overall *X* and *Y* range of the object (usually a model or scene object) that contains the items drawn in the object tree, accounting for any transformations of objects contained in the tree. Similarly, to find the appropriate near and far clipping planes, you can find the *Z* range of the object that contains the items drawn in the object tree. In practice, however, finding, adding, and transforming the ranges for a large object tree can be complicated.

Example Code

Two routines contained in the IDL distribution provide an example of how the view volume can be computed in many cases. These routines are defined in the files `set_view.pro` and `get_bounds.pro`, located in the `examples/doc/utilities` subdirectory of the IDL distribution. Run these example procedures by entering `set_view` or `get_bounds` at the IDL command prompt or view the files in an IDL Editor window by entering `.EDIT set_view.pro` or `.EDIT get_bounds.pro`.

The SET_VIEW procedure accepts as arguments the object references of a view object and a destination object, computes an appropriate view volume for the view object, and sets the VIEWPLANE_RECT property of the view object accordingly. The SET_VIEW procedure calls the GET_BOUNDS procedure to compute the *X*, *Y*, and *Z* ranges of the objects contained in the view object.

The SET_VIEW and GET_BOUNDS routines are used in the examples in this volume, and are available for your use when creating and displaying object hierarchies. They are, however, example code, and are not truly generic in the situations they address. When you encounter a situation for which these routines do not produce the desired result, we encourage you to copy and alter the code to suit your own needs.

Inspect the `SET_VIEW.PRO` and `GET_BOUNDS.PRO` files for further details.

Converting Data to Normal Coordinates

Most transformations are handled by the transformation matrix of a model object. For convenience, however, visualization objects may also have a simplified transformation applied to them. Coordinate transformations applied to individual graphic visualization objects allow you to change only the translation (position) and scale; this is useful when converting from one coordinate system to another.

For example, you may build your view object using normalized coordinates, so that values range between zero and one. If you create a graphic object—a surface object, say—based on the range of data values, you would need to convert your surface object (built with a data coordinate system) to match the view object (built with a normal coordinate system). To do this, use the [XYZ]COORD_CONV keywords to the graphic object in question. The [XYZ]COORD_CONV keywords take as their argument a two-element vector that specifies the translation and scale factor for each dimension.

Suppose you have a surface object whose data is specified in a range from [0, 0, *zMin*] to [*xMax*, *yMax*, *zMax*]. If you wanted to work with this surface as if it were in a normalized [–1, –1, –1] to [1, 1, 1] space, you could use the following coordinate conversions:

```
; Create some data:
myZdata = DIST(60)
; Use SIZE to determine size of each dimension of myZdata:
sz = SIZE(myZdata)
; Create a scale factor for the X dimension:
xs = 2.0/(sz[1]-1)
; Create a scale factor for the Y dimension:
ys = 2.0/(sz[2]-1)
; Create a scale factor for the Z dimension:
zs = 2.0/MAX(myZdata)
```

Now, use the [XYZ]COORD_CONV keywords to the IDLgrSurface::Init method to translate the surface by minus one unit in each direction, and to scale the surface by the scale factors:

```
mySurface = OBJ_NEW('IDLgrSurface', myZdata, $
   XCOORD_CONV = [-1, xs], YCOORD_CONV = [-1, ys], $
   ZCOORD_CONV = [-1, zs])
```

Remember that using the [XYZ]COORD_CONV keywords is simply a convenience—the above example could also have been written as follows:

```
; Create some data:
myZdata = DIST(60)
; Use SIZE to determine the size of each dimension of myZdata:
```

```
sz = SIZE(myZdata)
; Create a scale factor for the X dimension:
xs = 2.0/(sz(1)-1)
; Create a scale factor for the Y dimension:
ys = 2.0/(sz(2)-1)
; Create a scale factor for the Z dimension:
zs = 2.0/(MAX(myZdata)
; Create a model object:
myModel = OBJ_NEW('IDLgrModel')
; Apply scale factors:
myModel->Scale, xs, ys, zs
; Translate:
myModel->Translate, -1, -1, -1
; Create surface object:
mySurface = OBJ_NEW('IDLgrSurface', myZdata)
; Add surface object to model object:
myModel->Add, mySurface
```

A Function for Coordinate Conversion

Often, it is convenient to convert minimum and maximum data values so that they fit in the range from 0.0 to 1.0 (that is, so they are normalized). Rather than adding the code to make this coordinate conversion to your code in each place it is required, you may wish to define a coordinate conversion function.

For example, the following function definition accepts a two-element array representing minimum and maximum values returned by the XYZRANGE keyword to the GetProperty method, and returns two-element array of scaling parameters suitable for the XYZCOORD_CONV keywords:

```
FUNCTION NORM_COORD, range
   scale = [-range[0]/(range[1]-range[0]), 1/(range[1]-range[0])]
   RETURN, scale
END
```

If you define a function like this in your code, you can then call it whenever you need to scale your data ranges into normalized coordinates. The following statements create a plot object from the variable data, retrieve the values of the *X* and *Y* ranges for the plot, and the use the XYCOORD_CONV keywords to the SetProperty method and the NORM_COORD function to set the coordinate conversion.

```
plot = OBJ_NEW('IDLgrPlot', data)
plot->GetProperty, XRANGE=xr, YRANGE=yr
plot->SetProperty, XCOORD_CONV=NORM_COORD(xr), $
   YCOORD_CONV=NORM_COORD(yr)
```

Example Code

The function NORM_COORD is defined in the file `norm_coord.pro` in the `examples/doc/utilities` subdirectory of the IDL distribution. Run this example procedure by entering `norm_coord` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT norm_coord.pro`.

Example: Centering an Image

The following example steps through the process of creating an image object and provides two options for centering it within a window.

The first method establishes a viewplane rectangle within a view object. The image object is added to a model object. The model object is then translated to the center of the window object.

The second method does not establish a viewplane rectangle. Instead coordinate conversions are calculated and applied to the image object to center it within the model. This method works within the normalized coordinate system of the model.

You can also position an image in a view using the LOCATION property of the image object. For additional information and examples, see “Positioning Image Objects in a View” on page 105.

This example uses the image from the `worldelv.dat` file found in the `examples/data` directory.

```
PRO CenteringAnImage

; Determine path to file.
worldelvFile = FILEPATH('worldelv.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize image parameters.
worldelvSize = [360, 360]
worldelvImage = BYTARR(worldelvSize[0], worldelvSize[1], /NOZERO)

; Open file, read in image, and close file.
OPENR, unit, worldelvFile, /GET_LUN
READU, unit, worldelvImage
FREE_LUN, unit

; Initialize window parameters.
windowSize = [400, 460]
windowMargin = (windowSize - worldelvSize)/2

; First Method:  Defining the Viewplane and
;                Translating the Model.
;-------------------------------------------
```

```
; Initialize objects required for an Object Graphics
; display.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = windowSize, $
   TITLE = 'World Elevation:  First Method')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., windowSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize palette with STD GAMMA-II color table and
; use it to initialize the image object.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette->LOADCT, 5
oImage = OBJ_NEW('IDLgrImage', worldelvImage, PALETTE = oPalette)

; Add image to model, which is added to view.  Model
; is translated to center the image within the window.
; Then view is displayed in window.
oModel->Add, oImage
oView->Add, oModel
oModel->Translate, windowMargin[0], windowMargin[1], 0.
oWindow->Draw, oView

; Clean-up object references.
OBJ_DESTROY, [oView, oPalette]

; Second Method:  Using Coordinate Conversions.
;-----------------------------------------------

; Initialize objects required for an Object Graphics
; display.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = windowSize, $
   TITLE = 'World Elevation:  Second Method')
oView = OBJ_NEW('IDLgrView')
oModel = OBJ_NEW('IDLgrModel')

; Initialize palette with STD GAMMA-II color table and
; use it to initialize the image object.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette->LOADCT, 5
oImage = OBJ_NEW('IDLgrImage', worldelvImage, $
   PALETTE = oPalette)

; Obtain initial coordinate conversions of image object.
oImage->GetProperty, XCOORD_CONV = xConv, $
   YCOORD_CONV = yConv, XRANGE = xRange, YRANGE = yRange
```

```
; Output initial coordinate conversions.
PRINT, 'Initial xConv:  ', xConv
PRINT, 'Initial yConv:  ', yConv

; Applying margins to coordinate conversions.
xTranslation = (2.*FLOAT(windowMargin[0])/windowSize[0]) - 1.
xScale = (-2.*xTranslation)/worldelvSize[0]
xConv = [xTranslation, xScale]
yTranslation = (2.*FLOAT(windowMargin[1])/windowSize[1]) - 1.
yScale = (-2.*yTranslation)/worldelvSize[1]
yConv = [yTranslation, yScale]

; Output resulting coordinate conversions.
PRINT, 'Resulting xConv:  ', xConv
PRINT, 'Resulting yConv:  ', yConv

; Apply resulting conversions to the image object.
oImage->SetProperty, XCOORD_CONV = xConv, $
   YCOORD_CONV = yConv

; Add image to model, which is added to view. Display
; the view in the window.
oModel->Add, oImage
oView->Add, oModel
oWindow->Draw, oView

; Cleanup object references.
OBJ_DESTROY, [oView, oPalette]

END
```

Example: Transforming a Surface

The following example steps through the process of creating a surface object and all of the supporting objects necessary to display it.

Example Code

The procedure file `test_surface.pro`, located in the `examples/doc/objects` subdirectory of the IDL distribution, contains this example's code. Run this example procedure by entering `test_surface` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT test_surface.pro`.

When creating this procedure, we allow the user to specify keywords that will return object references to the view, model, surface, and window objects. This allows us to manipulate the objects directly from the IDL command line after the procedure has been run.

Play with the example to learn how object transformations work and interact. Try the following commands at the IDL prompt to observe what they do:

First, compile `test_surface.pro`:

```
.RUN test_surface.pro
```

Now, execute the procedure. The variables you supply via the SURFACE, MODEL, VIEW, and WINDOW keyword will contain object references you can manipulate from the command line:

```
test_surface, VIEW=myview, MODEL=mymodel, $
   SURFACE=mysurf, WINDOW=mywin
```

This will create a window object and display the surface. Now try the following to translate the object to the right:

```
mymodel->Translate, 0.2, 0, 0
```

The model transformation changes as soon as you issue this command. The window object, however, will not be updated to reflect the new position until you issue a Draw command:

```
mywin->Draw, myview
```

Try a rotation in the *y* direction:

```
mymodel->Rotate, [0,1,0], 45
mywin->Draw, myview
```

Repeat the commands several times and observe what happens.

Try some of the following. Remember to issue a Draw command after each change in order to see what you have done.

```
mymodel->Scale, 0.5, 0.5, 0.5
mymodel->Scale, 1, 0.5, 1
mymodel->Scale, 1, 2, 1
mymodel->Rotate, [0,0,1], 45
mysurf->SetProperty, COLOR = [0, 255, 0]
myview->SetProperty, PROJECTION = 2, EYE = 2
myview->SetProperty, EYE = 1.1
myview->SetProperty, EYE = 6
```

Zooming within an Object Display

Enlarging a specific section of an image is known as zooming. How zooming is performed within IDL depends on the graphics system. In Direct Graphics, you can use the ZOOM procedure to zoom in on a specific section of an image. If you are working with RGB images, you can use the ZOOM_24 procedure.

In Object Graphics, the VIEWPLANE_RECT keyword is used to change the view object. Using this method, the entire image is still contained within the image object, while the view is changed to only show specific areas of the image object. See the following section for more information.

Zooming in on an Object Graphics Image Display

The following example imports a grayscale image from the `convec.dat` binary file. This grayscale image shows the convection of the Earth's mantle. The image contains byte data values and is 248 pixels by 248 pixels. The VIEWPLANE_RECT keyword to the view object is updated to zoom in on the lower left corner of the image.

Example Code

See `zooming_object.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for code that duplicates this example. Run this example procedure by entering `zooming_object` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT zooming_object.pro`.

1. Determine the path to the `convec.dat` file:

```
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Initialize the image size parameter:

```
imageSize = [248, 248]
```

3. Import the image from the file:

```
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

4. Initialize the display objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Grayscale Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')
```

5. Initialize the image object:

```
oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)
```

6. Add the image object to the model, which is added to the view, then display the view in the window:

```
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView
```

The following figure shows the resulting grayscale image display.

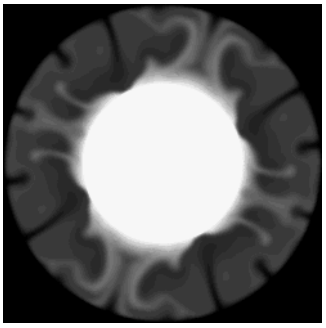


Figure 3-6: A Grayscale Image in Object Graphics

7. Initialize another window:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'Zoomed Image')
```

8. Change the view to enlarge the lower left quarter of the image:

```
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize/2]
```

The view object still contains the entire image object, but the region displayed by the view (the viewplane rectangle) is reduced in size by half in both directions. Since the window object remains the same size, the view region is enlarged to fit it to the window.

9. Display the updated view in the new window:

```
oWindow -> Draw, oView
```

The following figure shows the resulting zoomed image.

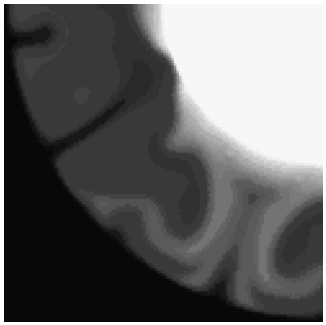


Figure 3-7: Enlarged Image Area in Object Graphics

10. Clean up the object references. When working with objects always remember to clean up any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

```
OBJ_DESTROY, oView
```

Translating, Rotating and Scaling Objects

An IDLgrModel object is a container for any visualization objects that are to be rotated, translated, or scaled. Each IDLgrModel object has a transformation property (set via the TRANSFORM keyword to the IDLgrModel::Init or SetProperty method), which is a 4 x 4 floating-point matrix. For a general discussion of transformation matrices and three-dimensional graphics, see "Coordinates of 3-D Graphics" (Chapter 5, *Using IDL*).

Note

A model object's transformation matrix is akin to the transformation matrix used by IDL Direct Graphics and stored in the !P.T system variable field. Transformation matrices associated with a model object do not use the value of !P.T, however, and are not affected by the T3D procedure used in Direct Graphics.

By default, a model object's transformation matrix is set equal to a 4-by-4 identity matrix:

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Figure 3-8:

You can change the transformation matrix of a model object directly, using the TRANSFORM keyword to the IDLgrModel::Init or SetProperty method:

```
myModel = OBJ_NEW('IDLgrModel', TRANSFORM = tmatrix)
```

where *tmatrix* is a 4-by-4 transformation matrix. Alternatively, you can use the Translate, Rotate, and Scale methods to the IDLgrModel object to alter the model's transformation matrix.

Translation

The IDLgrModel::Translate method takes three arguments specifying the amount to translate the model object and its contents in the *X*, *Y*, and *Z* directions. For example, to translate a model and its contents by 1 unit in the *X*-direction, you could use the following statements:

```
dx = 1 & dy = 0 & dz = 0
myModel->Translate, dx, dy, dz
```

How does this affect the transformation matrix? Notice that we could change the transformation matrix in an identical way using the following statements:

```
; Define translation values:
dx = 1 & dy = 0 & dz = 0
; Get existing transformation matrix:
myModel->GetProperty, TRANSFORM = oldT
; Provide a transformation matrix that performs the translation:
transT = [[1.0, 0.0, 0.0, dx], $
          [0.0, 1.0, 0.0, dy], $
          [0.0, 0.0, 1.0, dz], $
          [0.0, 0.0, 0.0, 1.0]]
; Multiply the existing transformation matrix by
; the matrix that performs the translation:
newT = oldT # transT
; Apply the new transformation matrix to the model object:
myModel->SetProperty, TRANSFORM = newT
```

Rotation

The IDLgrModel::Rotate method takes two arguments specifying the axis about which to rotate and the number of degrees to rotate the model object and its contents. For example, to rotate a model and its contents by 90 degrees around the *y*-axis, you could use the following statements:

```
axis = [0,1,0] & angle = 90
myModel->Rotate, axis, angle
```

How does this affect the transformation matrix? Notice that we could change the transformation matrix in an identical way using the following statements:

```
; Define rotation values:
axis = [0,1,0] & angle = 90
; Get existing transformation matrix:
myModel->GetProperty, TRANSFORM = oldT
; Define sine and cosine of angle:
cosa = COS(!DTOR*angle)
sina = SIN(!DTOR*angle)
```

```
; Provide a transformation matrix that performs the rotation:
rotT = [[cosa, 0.0, sina, 0.0], $
        [0.0, 1.0, 0.0, 0.0], $
        [-sina, 0.0, cosa, 0.0], $
        [0.0, 0.0, 0.0, 1.0]]
; Multiply the existing transformation matrix
; by the matrix that performs the rotation.
newT = oldT # rotT
; Apply the new transformation matrix to the model object:
myModel->SetProperty, TRANSFORM = newT
```

Scaling

The IDLgrModel::Scale method takes three arguments specifying the amount to scale the model object and its contents in the *x*, *y*, and *z* directions. For example, to scale a model and its contents by 2 units in the *y* direction, you could use the following statements:

```
sx = 1 & sy = 2 & sz = 1
myModel->Scale, sx, sy, sz
```

How does this affect the transformation matrix? Notice that we could change the transformation matrix in an identical way using the following statements:

```
; Define scaling values:
sx = 1 & sy = 2 & sz = 1
; Get existing transformation matrix:
myModel->GetProperty, TRANSFORM = oldT
; Provide a transformation matrix that performs the scaling:
scaleT = [[sx, 0.0, 0.0, 0.0], $
          [0.0, sy, 0.0, 0.0], $
          [0.0, 0.0, sz, 0.0], $
          [0.0, 0.0, 0.0, 1.0]]
; Multiply the existing transformation matrix
; by the matrix that performs the scaling.
newT = oldT # scaleT
; Apply the new transformation matrix to the model object:
myModel->SetProperty, TRANSFORM = newT
```

Combining Transformations

Note that model transformations are cumulative. That is, a model object contained in another model is subject to both its own transformation and to that of its container. All transformation matrices that apply to a given model object are multiplied together when the object is rendered. For example, consider a model that contains another model:

```
model1 = OBJ_NEW('IDLgrModel', TRANSFORM = trans1)
model2 = OBJ_NEW('IDLgrModel', TRANSFORM = trans2)
model2->Add, model1
```

The model1 object is now subject to both its own transformation matrix (`trans1`) and to that of its container (`trans2`). The result is that when `model1` is rendered, it will be rendered with a transformation matrix = `trans1` # `trans2`.

Interactive 3D Transformations

To create truly interactive object graphics, you must allow the user to transform the position or orientation of objects using the mouse. One way to do this is to provide a virtual trackball that lets the user manipulate objects interactively on the screen.

Note

The iTools provide extensive interactivity for all types of object data displayed in an iTool. This interactivity is automatically available when suitable data is displayed in an iTool. See the *iTool User's Guide* for complete details.

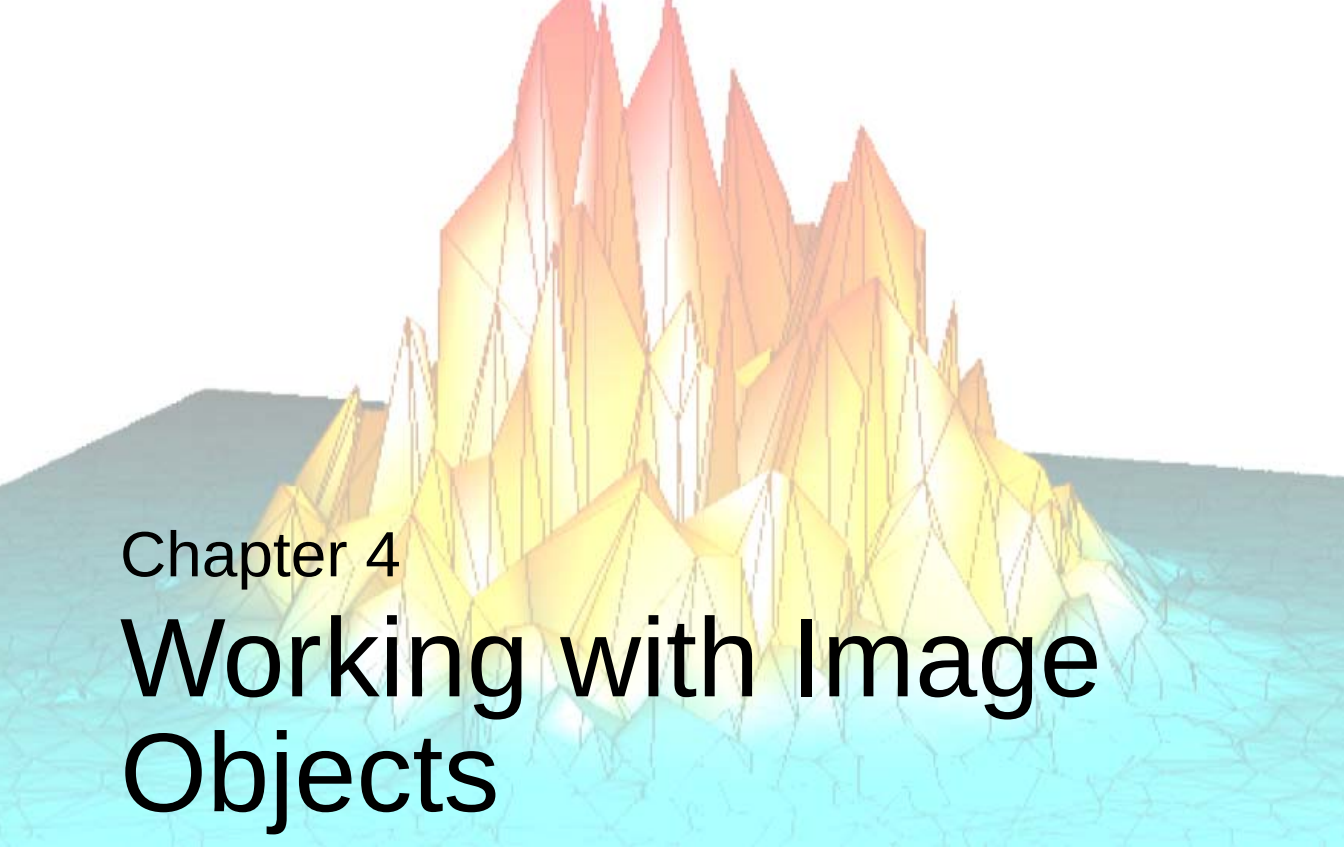
The procedure file `trackball__define.pro`, found in the `lib` directory of the IDL distribution, contains the object definition procedure for a virtual trackball object. This trackball object is used in several of the examples presented in this volume, and is also used by other example and demonstration code included with IDL. The trackball object has three methods: Init, Update, and Reset. These methods allow you to retrieve mouse movement events and alter your model transformations accordingly.

The trackball object behaves as if there were an invisible trackball, centered at a position you specify, overlaid on a draw widget. The widget application's event handler uses the widget event information to update both the trackball's state and the model transformation of the objects displayed in the draw widget's window object. When the user clicks and drags in the draw widget, objects in the draw widget rotate as if the user were manipulating them with a physical trackball.

See "TrackBall" (*IDL Reference Guide*) for details on creating and using trackball objects. Several of the other example files located in the `examples/doc/objects` subdirectory of the IDL distribution include trackball objects, and may be studied for further insight into the mechanics of transforming object hierarchies based on user input.

Note

The XOBJVIEW procedure is a utility used to quickly and easily view and manipulate IDL Object Graphics on screen. Pre-built functionality allows you to select, rotate, pan and zoom objects contained within the model(s) passed to the procedure. See "XOBJVIEW" (*IDL Reference Guide*) for details.



Chapter 4

Working with Image Objects

The following topics are covered in this chapter:

Overview of Image Objects

An object of the IDLgrImage class (see "IDLgrImage" (*IDL Reference Guide*)) represents a two-dimensional array of pixel values, rendered on the plane z = 0. The image object stores image data using the byte data type, and can take any of the following forms:

- An array with dimensions [n, m]. Each pixel is interpreted as an index into a palette, or as an explicit gray scale value (if the GREYSCALE keyword is set).
- An array with dimensions [2, n, m] or [n, 2, m] or [n, m, 2]. Each pixel consists of a gray scale value and an associated alpha channel value (alpha is used for transparency effects).
- An array with dimensions [3, n, m] or [n, 3, m] or [n, m, 3]. Each pixel consists of an RGB triple.
- An array with dimensions [4, n, m] or [n, 4, m] or [n, m, 4]. Each pixel consists of an RGB triple and an associated alpha channel value.

The index or RGB triple for each pixel is interpreted according to the color model set for the destination object in which it is to be drawn. The Alpha channel, if present, determines the transparency of the pixel.

Note

The position of the color bands in an RGB image array is know as interleaving. See "RGB Image Interleaving" (Chapter 5, *Using IDL*) for details. The INTERLEAVE property of the image object describes this arrangement.

Defining Image Palettes

If your image array contains indexed color data (that is, if it is an *m*-by-*n* array), you can specify a palette object to control the conversion between the image data and the palette used by an RGB-mode destination object. (See "How IDL Interprets Color Values" on page 53 for a discussion of the interaction between indexed color objects and RGB color destinations.) Set the PALETTE property of the image object equal to an instance of an IDLgrPalette object:

```
myimage->SetProperty, PALETTE = mypalette
```

To specify that an image be drawn in greyscale mode rather than through an existing color palette, set the GREYSCALE property equal to 1 (one). The GREYSCALE property is only used if the image data is a single channel (an *m*-by-*n* array).

Note

A 2-by-*m*-by-*n* array is considered to be a greyscale image with an Alpha channel. An image containing indexed color data cannot have an alpha channel.

For examples, see "Displaying Indexed Images with Object Graphics" in the Examples section of "IDLgrPalette" (*IDL Reference Guide*).

Configuring Common Object Properties

IDLgrImage properties allow you to configure how image objects are displayed. You can alter the transparency (using the ALPHA_CHANNEL keyword), or the color (using the PALETTE keyword for indexed images, or the INTERLEAVE keyword for RGB images). You may want to fit one image to another using warping or create a texture map by mapping an image onto a geometric shape. See the following sections for more information.

- "Creating Image Objects" on page 100 provides examples and resources for creating image objects containing a variety of data
- "Positioning Image Objects in a View" on page 105
- "Defining Transparency in Image Objects" on page 115
- "Warping Image Objects" on page 121
- "Mapping an Image Object onto a Sphere" on page 132

If you want to display very large images, you can do so with image tiling. See "Image Tiling" on page 136 for information.

Creating Image Objects

To create an image object, supply an array of pixel values to the IDLgrImage::Init method. If the image has more than one channel, be sure to set the INTERLEAVE property of the image object to the appropriate value. (See "RGB Image Interleaving" (Chapter 5, *Using IDL*) for details and an example showing how to determine the interleaving within an image array.) See "IDLgrImage" (*IDL Reference Guide*) for details on object properties and methods.

Note

IDLgrImage does not treat NaN data as missing. If the image data includes NaNs, it is recommended that the BYTSCL function be used to appropriately handle those values. For example:

```
oImage->SetProperty, DATA = BYTSCL(myData, /NaN, MIN=0,
   MAX=255)
```

In Object Graphics, binary, grayscale, indexed, and RGB images are contained in image objects. For display, the image object is contained within an object hierarchy, which includes a model object and a view object. The view object is then drawn to a window object. Some types of images must be scaled with the BYTSCL function prior to display.

For more information, refer to the following examples:

- "Displaying Binary Images with Object Graphics" below
- "Displaying Grayscale Images with Object Graphics" on page 102
- "Displaying Indexed Images with Object Graphics" in the Examples section of "IDLgrPalette" (*IDL Reference Guide*).
- "RGB Image Interleaving" (Chapter 5, *Using IDL*)

Displaying Binary Images with Object Graphics

Binary images are composed of pixels having one of two values, usually zero or one. With most color tables, pixels having values of zero and one are displayed with almost the same color, such as with the default grayscale color table. Thus, a binary image is usually scaled to display the zeros as black and the ones as white.

The following example imports a binary image of the world from the `continent_mask.dat` binary file. In this image, the oceans are zeros (black) and the continents are ones (white). This type of image can be used to mask out (omit)

data over the oceans. The image contains byte data values and is 360 pixels by 360 pixels.

Example Code

See `displaybinaryimage_object.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for code that duplicates this example. Run the example procedure by entering `displaybinaryimage` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT displaybinaryimage.pro`.

1. Determine the path to the `continent_mask.dat` file:

```
file = FILEPATH('continent_mask.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Initialize the image size parameter:

```
imageSize = [360, 360]
```

3. Use READ_BINARY to import the image from the file:

```
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

4. Initialize the display objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Binary Image, Not Scaled')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')
```

5. Initialize the image object:

```
oImage = OBJ_NEW('IDLgrImage', image)
```

6. Add the image object to the model, which is added to the view, then display the view in the window:

```
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView
```

   The resulting window should be all black (blank). The binary image contains zeros and ones, which are almost the same color (black). A binary image should be scaled prior to displaying in order to show the ones as white.

7. Initialize another window:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
```

```
    TITLE = 'A Binary Image, Scaled')
```

8. Update the image object with a scaled version of the image:

```
oImage -> SetProperty, DATA = BYTSCL(image)
```

9. Display the view in the window:

```
oWindow -> Draw, oView
```

The following figure shows the results of scaling this display.

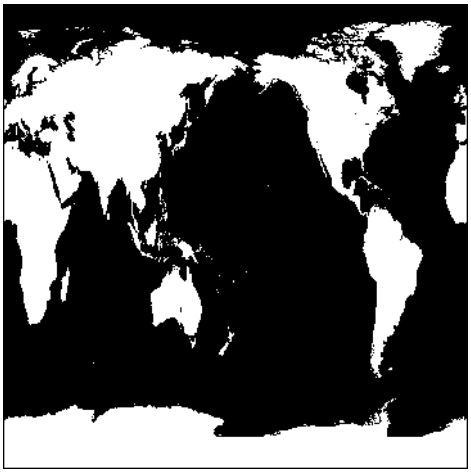


Figure 4-1: Binary Image in Object Graphics

10. Clean up the object references. When working with objects always remember to clean up any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

```
OBJ_DESTROY, oView
```

Displaying Grayscale Images with Object Graphics

Since grayscale images are composed of pixels of varying intensities, they are best displayed with color tables that progress linearly from black to white. IDL provides several such pre-defined color tables, but the default grayscale color table is generally suitable.

The following example imports a grayscale image from the `convec.dat` binary file. This grayscale image shows the convection of the Earth's mantle. The image contains byte data values and is 248 pixels by 248 pixels. Since the data type is byte, this image does not need to be scaled before display. If the data was of any type other than byte and the data values were not within the range of 0 up to 255, the display would need to scale the image in order to show its intensities. Complete the following steps for a detailed description of the process.

Example Code

See `displaygrayscaleimage_object.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for code that duplicates this example. Run the example procedure by entering `displaygrayscaleimage` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT displaygrayscaleimage.pro`.

1. Determine the path to the `convec.dat` file:

```
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Initialize the image size parameter:

```
imageSize = [248, 248]
```

3. Using READ_BINARY, import the image from the file:

```
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

4. Initialize the display objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Grayscale Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')
```

5. Initialize the image object:

```
oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)
```

6. Add the image object to the model, which is added to the view, then display the view in the window:

```
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView
```

The following figure shows the resulting grayscale image display.

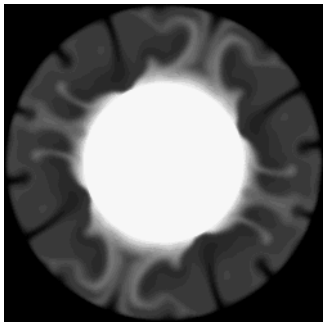


Figure 4-2: Grayscale Image in Object Graphics

7. Clean up the object references. When working with objects always remember to clean up any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

```
OBJ_DESTROY, oView
```

Positioning Image Objects in a View

By default, IDLgrImage objects are drawn at Z=0 and are positioned and sized with respect to two points:

```
p1 = [LOCATION(0), LOCATION(1), 0]
p2 = [LOCATION(0) + DIMENSION(0), LOCATION(1) + DIMENSION(1), 0].
```

where LOCATION and DIMENSION are properties of the image object. These two points are transformed in three dimensions, and then projected onto the screen to form the opposite corners of a 2-D rectangle resulting in screen space points designated as `p1'` and `p2'`:

```
[ [p1'[0], p1'[1]], [[p2'[0], p1'[1]],
   [[p2'[0], p2'[1]], [[p1'[0], p2'[1]] ]
```

The image data is drawn on the display as a 2-D image within this 2-D rectangle whose sides are parallel to the screen sides. The image data is *scaled* in two dimensions (not rotated) to fit into this projected rectangle and then drawn with Z buffering disabled.

To draw an image with the current full 3D transformation (the same way other objects such as polygons are transformed), set the IDLgrImage TRANSFORM_MODE property to 1. See the IDLgrImage TRANSFORM_MODE property in the *IDL Reference Guide* for details.

Objects are drawn to a destination device in the order that they are added (via the Add method) to the model, view, or scene that contains them. By default, image objects do not take into account the depth locations of other objects that may be included in the view object unless you enable depth testing (see “DEPTH_TEST_DISABLE” (*IDL Reference Guide*) for details).

This means that objects that are drawn to the destination object (window or printer) after the image is drawn will appear to be in front of the image, even if they are located behind the image object. And this also means that objects drawn after the image is drawn will appear to be in front of the image even if they are located behind the image. Since the image is drawn by default with depth testing disabled, you can think of the image primitive as ‘painting’ the image onto the screen without regard for other objects that might already have been drawn there.

This behavior can be changed by enabling depth testing to make the image primitive behave like other primitives such as polygons when they are drawn with depth testing enabled.

The following example uses the LOCATION keyword to control image position. For information on other ways to define the position of an image object in a view, see "Example: Centering an Image" on page 83.

Displaying Multiple Images in Object Graphics

The following example imports an RGB image from the `rose.jpg` image file. This RGB image is a close-up photograph of a red rose and is pixel interleaved. This example extracts the three color channels of this image, and displays them as grayscale images in various locations within the same window. Complete the following steps for a detailed description of the process.

Example Code

See `displaymultiples_object.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for code that duplicates this example. Run the example procedure by entering `displaymultiples_object` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT displaymultiples_object.pro`.

1. Determine the path to the `rose.jpg` file:

```
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Use QUERY_IMAGE to query the file to determine image parameters:

```
queryStatus = QUERY_IMAGE(file, imageInfo)
```

3. Set the image size parameter from the query information:

```
imageSize = imageInfo.dimensions
```

4. Use READ_IMAGE to import the image from the file:

```
image = READ_IMAGE(file)
```

5. Extract the channels (as images) from the pixel interleaved RGB image:

```
redChannel = REFORM(image[0, *, *])
greenChannel = REFORM(image[1, *, *])
blueChannel = REFORM(image[2, *, *])
```

The LOCATION keyword to the Init method of the image object can be used to position an image within a window. The LOCATION keyword uses data coordinates, which are the same as device coordinates for images. Before initializing the image objects, you should initialize the display objects. The following steps display multiple images horizontally, vertically, and diagonally.

6. Initialize the display objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[3, 1], $
   TITLE = 'The Channels of an RGB Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 3, 1])
oModel = OBJ_NEW('IDLgrModel')
```

7. Now initialize the image objects and arrange them with the LOCATION keyword, see IDLgrImage for more information:

```
oRedChannel = OBJ_NEW('IDLgrImage', redChannel)
oGreenChannel = OBJ_NEW('IDLgrImage', greenChannel, $
   LOCATION = [imageSize[0], 0])
oBlueChannel = OBJ_NEW('IDLgrImage', blueChannel, $
   LOCATION = [2*imageSize[0], 0])
```

8. Add the image objects to the model, which is added to the view, then display the view in the window:

```
oModel -> Add, oRedChannel
oModel -> Add, oGreenChannel
oModel -> Add, oBlueChannel
oView -> Add, oModel
oWindow -> Draw, oView
```

The following figure shows the resulting grayscale images.

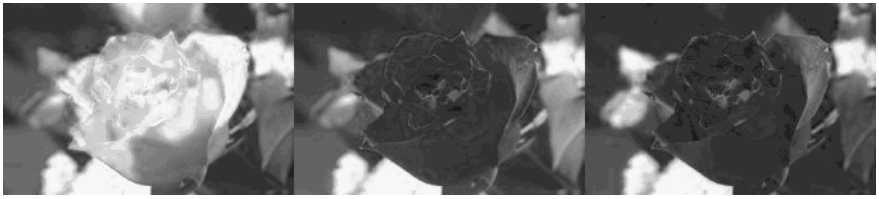


Figure 4-3: Horizontal Display of RGB Channels in Object Graphics

These images can be displayed vertically in another window by first initializing another window and then updating the view and images with different location information.

9. Initialize another window object:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[1, 3], $
   TITLE = 'The Channels of an RGB Image')
```

10. Change the view from horizontal to vertical:

```
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 1, 3]
```

11. Change the locations of the channels:

```
oGreenChannel -> SetProperty, LOCATION = [0, imageSize[1]]
oBlueChannel -> SetProperty, LOCATION = [0, 2*imageSize[1]]
```

12. Display the updated view within the new window:

```
oWindow -> Draw, oView
```

The following figure shows the resulting grayscale images.

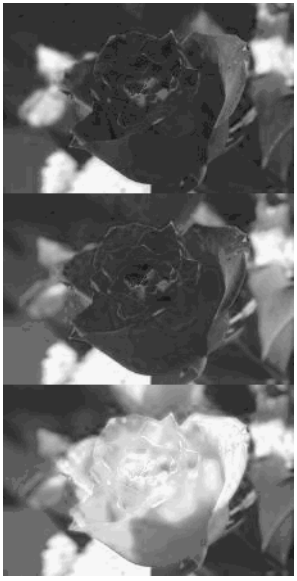


Figure 4-4: Vertical Display of RGB Channels in Object Graphics

These images can also be displayed diagonally in another window by first initializing the other window and then updating the view and images with different location information.The LOCATION can also be used to create a display overlapping images. When overlapping images in Object Graphics, you must remember the last image added to the model will be in front of the previous images.

13. Initialize another window object:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[2, 2], $
   TITLE = 'The Channels of an RGB Image')
```

14. Change the view to prepare for a diagonal display:

```
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 2, 2]
```

15. Change the locations of the channels:

```
oGreenChannel -> SetProperty, $
   LOCATION = [imageSize[0]/2, imageSize[1]/2]
oBlueChannel -> SetProperty, $
   LOCATION = [imageSize[0], imageSize[1]]
```

16. Display the updated view within the new window:

```
oWindow -> Draw, oView
```

The following figure shows the resulting grayscale images.

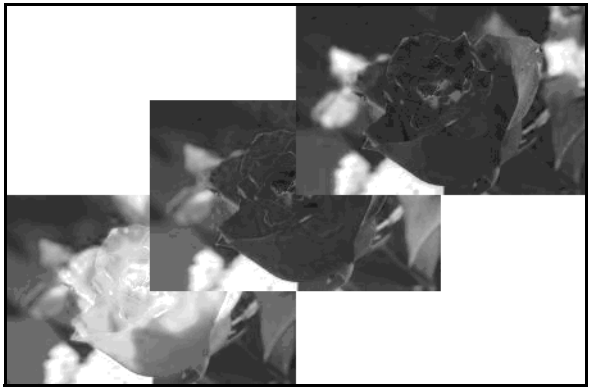


Figure 4-5: Diagonal Display of RGB Channels in Object Graphics

17. Clean up the object references. When working with objects always remember to clean up any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

```
OBJ_DESTROY, oView
```

Panning in Object Graphics

In Object Graphics, the VIEWPLANE_RECT keyword is used to change the view object. The entire image is still contained within the image object, but the view is changed to pan over specific areas of the image object.

The following example imports a grayscale image from the `nyny.dat` binary file. This grayscale image is an aerial view of New York City. The image contains byte data values and is 768 pixels by 512 pixels. The VIEWPLANE_RECT keyword to the view object is updated to zoom in on the lower left corner of the image. Then the VIEWPLANE_RECT keyword is used to pan over the bottom edge of the image. Complete the following steps for a detailed description of the process.

Example Code

See `panning_object.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for code that duplicates this example. Run the example procedure by entering `panning_object` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT panning_object.pro`.

1. Determine the path to the `nyny.dat` file:

```
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Initialize the image size parameter:

```
imageSize = [768, 512]
```

3. Import the image from the file:

```
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

4. Resize this large image to entirely display it on the screen:

```
imageSize = [256, 256]
image = CONGRID(image, imageSize[0], imageSize[1])
```

5. Initialize the display objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Grayscale Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')
```

6. Initialize the image object:

```
oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)
```

7. Add the image object to the model, which is added to the view, then display the view in the window:

```
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView
```

The following figure shows the resulting grayscale image display.

Figure 4-6: A Grayscale Image Of New York in Object Graphics

8. Initialize another window:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'Panning Enlarged Image')
```

9. Change the view to zoom into the lower left quarter of the image:

```
viewplane = [0., 0., imageSize/2]
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize/2]
```

The view object still contains the entire image object, but the region displayed by the view (the viewplane rectangle) is reduced in size by half in both directions. Since the window object remains the same size, the view region is enlarged to fit it to the window.

10. Display the updated view in the new window:

```
oWindow -> Draw, oView
```

The following figure shows the resulting enlarged image area.

Figure 4-7: Enlarged Image Area of New York in Object Graphics

11. Pan the view from the left side of the image to the right side of the image:

```
FOR i = 0, ((imageSize[0]/2) - 1) DO BEGIN & $
   viewplane = viewplane + [1., 0., 0., 0.] & $
   oView -> SetProperty, VIEWPLANE_RECT = viewplane & $
   oWindow -> Draw, oView & $
ENDFOR
```

Note

The `&` after BEGIN and the `$` allow you to use the FOR/DO loop at the IDL command line. These `&` and `$` symbols are not required when the FOR/DO loop in placed in an IDL program as shown in `Panning_Object.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory.

The following figure shows the resulting enlarged image area panned to the right side.

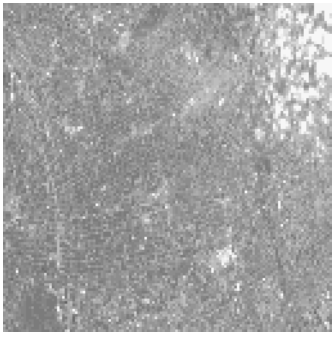


Figure 4-8: Enlarged New York Image Area Panned to the Right in Object Graphics

12. Clean up the object references. When working with objects always remember to clean up any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

```
OBJ_DESTROY, oView
```

Defining Transparency in Image Objects

In Object Graphics, a transparent image can be created by adding an alpha channel to the image array or by setting the ALPHA_CHANNEL property. The alpha channel is used to define the level of transparency in an image object. If you have an image containing both alpha channel data and a value for the ALPHA_CHANNEL property, the alpha values are combined by multiplying each image pixel alpha value by the ALPHA_CHANNEL property value.

If your image data includes an alpha channel, or if you set the ALPHA_CHANNEL property, use the BLEND_FUNCTION property of the image object to control how the alpha channel values will be interpreted. (See BLEND_FUNCTION property of IDLgrImage for details on how the blending is calculated.) This is known as alpha blending. For example, setting BLEND_FUNCTION = [3, 4] creates an image in which you can see through the foreground image to the background to the extent defined by the alpha channel values of the foreground image.

Transparency and Image Warping

Creating a transparent image is useful in the warping process when you want to overlay a transparency of the warped image onto the reference image (the image in which *Xo, Yo* control points were selected). See “Warping Image Objects” on page 121 for an example that uses transparent image objects.

For background information on warping images and selecting control points, see “Overview of Warping Images” (Chapter 5, *Image Processing in IDL*).

Image Transparency Examples

See the following topics for examples of creating transparent image objects:

- “Example: Applying a Transparent Image Overlay” on page 116 — layers two medical scan images of the brain. The opacity of the top image is controlled using the IDLgrImage ALPHA_CHANNEL property.
- “Example: Cumulative Alpha Blending” on page 118 — adds an alpha channel to an RGB image, masks out values, and then uses the ALPHA_CHANNEL property to control the image transparency.

Example: Applying a Transparent Image Overlay

The following example reads in two medical images, a computed tomography (CT) file that contains structural information, and a PET (positron emission tomography) file that contains metabolic data. A color table is applied to the PET file, and the transparency is set using the ALPHA_CHANNEL property. The PET image object is then overlaid on top of the base CT image. This is done by adding the transparent PET image to the model after (and therefore displayed in front of) the base CT image.

Example Code

See `alphaimage_obj_doc.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for code that duplicates this example. Run the example procedure by entering `alphaimage_obj_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT alphaimage_obj_doc.pro`.

To replicate this example, create a new `.pro` file and complete the following steps:

1. Load CT and PET images and get the image dimensions.

```
file_pt = FILEPATH('head_pt.dcm', $
   SUBDIRECTORY=['examples', 'data'])
file_ct = FILEPATH('head_ct.dcm', $
   SUBDIRECTORY=['examples', 'data'])
img_pt = READ_DICOM(file_pt)
img_ct = READ_DICOM(file_ct)
dims_ct = SIZE(img_ct, /DIMENSIONS)
dims_pt = SIZE(img_pt, /DIMENSIONS)
```

2. Check for dimension equality and resize if different.

```
IF dims_pt[0] NE dims_ct[0] THEN BEGIN
   x = dims_ct[0]/dims_pt[0]
   img_pt = REBIN(img_pt, dims_pt[0]*x, dims_pt[1]*x)
   dims_pt = x*dims_pt
   If dims_pt[0] NE dims_ct[0] THEN BEGIN
      status = DIALOG_MESSAGE ('Incompatible images', /ERROR)
   ENDIF
ENDIF
```

3. Change the data to byte type before creating the base CT image.

```
img_ct = BYTSCL(img_ct)
oImageCT = OBJ_NEW('IDLgrImage', img_ct)
```

4. Create display objects and display the CT image.

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN=2, $
   DIMENSIONS=[dims_ct[0], dims_ct[1]], TITLE='CT Image')
```

```
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0., 0., $
   dims_ct[0], dims_ct[1]])
oModel = OBJ_NEW('IDLgrModel')
oModel->Add, oImageCT
oView->Add, oModel
oWindow->Draw, oView
```

5. Create a palette object and load the red-temperature table.

```
oPalette = OBJ_NEW('IDLgrPalette')
oPalette->Loadct, 3
```

6. Change the data type to byte and create the PET image object. Set the BLEND_FUNCTION and ALPHA_CHANNEL properties to support image transparency.

```
img_pt = BYTSCL(img_pt)
oImagePT = OBJ_NEW('IDLgrImage', img_pt, $
   PALETTE=oPalette, BLEND_FUNCTION=[3,4], $
   ALPHA_CHANNEL=0.50)
```

7. Create a second window, add the semi-transparent image to the model containing the original image and display the overlay.

```
oWindow2 = OBJ_NEW('IDLgrWindow', RETAIN=2, $
   DIMENSIONS=[dims_pt[0], dims_pt[1]],  $
   LOCATION=[dims_ct[0]+10, 0], TITLE='CT/PET Transparency')
oModel -> Add, oImagePT
oWindow2 -> Draw, oView
```

8. Clean-up object references.

```
OBJ_DESTROY, [oView, oImageCT, oImagePT]
```

The results of this example are shown in the following figure.

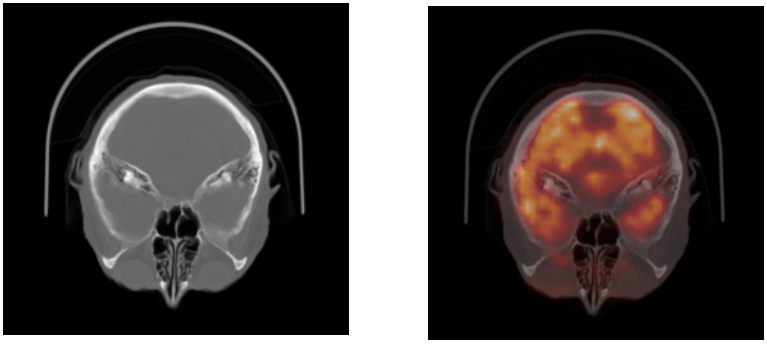


Figure 4-9: CT Image (Left) and CT with Semi-transparent PET Overlay (Right)

Example: Cumulative Alpha Blending

The following example shows the additive effects of displaying an image object with alpha channel data and an image with an ALPHA_CHANNEL property setting. In this example, the alpha channel is used to mask out values, and the ALPHA_CHANNEL property is used to control the object transparency. However, it is easy to modify the code and investigate the relationship between setting image transparency using the alpha channel data and ALPHA_CHANNEL property. For example, defining 50% transparency for each results in 25% opacity overall.

The two initial images are displayed in the following figure. The black portion of the land classification image (left) will be removed and this image will then be overlaid on top of the map image.

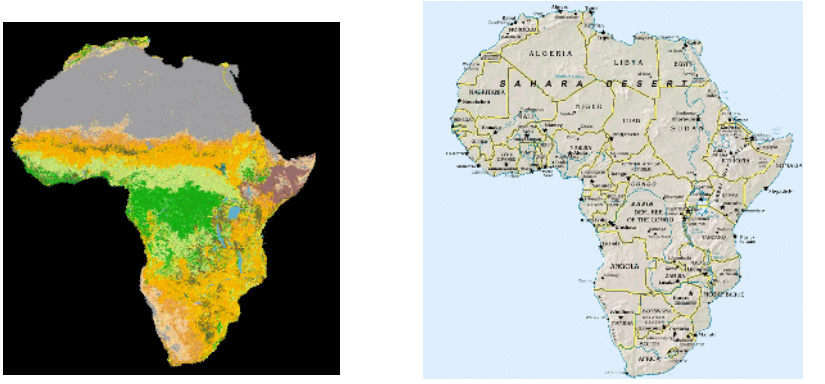


Figure 4-10: Original Land and Map Images

Example Code

See `alphacomposite_image_doc.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for code that duplicates this example. Run the example procedure by entering `alphacomposite_image` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT .EDIT alphacomposite_image.pro`.

To replicate this example, create a new `.pro` file complete the following steps:

1. Open the political map, the base image.

```
mapFile = FILEPATH('afrpolitsm.png', $
   SUBDIRECTORY = ['examples', 'data'])
mapImg = READ_PNG(mapFile, mapR, mapG, mapB)
```

2. Assign the color table of the map image to a palette object.

```
mapPalette = OBJ_NEW('IDLgrPalette', mapR, mapG, mapB)
```

3. Create an image object containing the map data.

```
oMapImg = OBJ_NEW('IDLgrImage', mapImg, $
   DIMENSIONS=[600, 600], PALETTE=mapPalette)
```

4. Open the land cover characteristics image.

```
landFile = FILEPATH('africavlc.png', $
   SUBDIRECTORY = ['examples', 'data'])
landImg = READ_PNG(landFile, landR, landG, landB)
landImgDims = SIZE(landImg, /DIMENSIONS)
```

5. To mask out the black values of the land classification image, create a 4 channel array for the red, green, blue, and alpha data.

```
alphaLand = BYTARR(4, landImgDims[0], landImgDims[1],$
   /NOZERO)
```

6. Get the red, green and blue values used by the image and assign them to the first three channels of the alpha image array.

```
alphaLand[0, *, *] = landR[landImg]
alphaLand[1, *, *] = landG[landImg]
alphaLand[2, *, *] = landB[landImg]
```

7. Mask out the black pixels with a value of 0. Multiply the mask value by 255 for complete opacity. You could set this to a value between 0 (completely transparent) and 255 (opaque) to control the transparency. Any value set here will be combined with any value set for the ALPHA_CHANNEL property on the image object.

```
mask = (landImg GT 0)
alphaLand [3, *, *] = mask*255B
```

8. Create the semi-transparent image object. ALPHA_CHANNEL values can range from 0.0 (transparent) to 1.0 (opaque). The image will appear semi-transparent when the BLEND_FUNCTION property is set to [3,4].

```
oAlphaLand = OBJ_NEW('IDLgrImage', alphaLand, $
   DIMENSIONS=[600, 600], BLEND_FUNCTION=[3,4], $
   ALPHA_CHANNEL=0.35)
```

9. Create the display objects.

```
oWindow = OBJ_NEW('IDLgrWindow', $
   DIMENSIONS=[600, 600], RETAIN=2, $
   TITLE='Overlay of Land Cover Transparency')
viewRect = [0, 0, 600, 600]
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=viewRect)
```

```
oModel = OBJ_NEW('IDLgrModel')
```

10. Add the semi-transparent image to the model after the base image.

```
oModel->Add, oMapImg
oModel->Add, oAlphaLand
oView->Add, oModel
oWindow->Draw, oView
```

11. Clean up objects.

```
OBJ_DESTROY, [oView, oMapImg, oAlphaLand, mapPalette]
```

The results appear in the following figure.

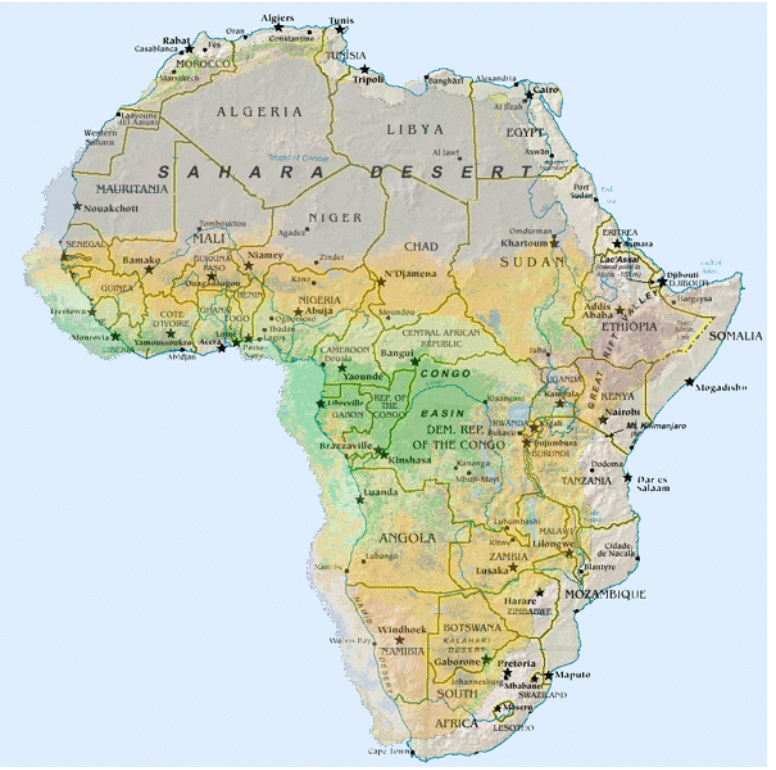


Figure 4-11: Land Image (35% Opaque) Overlaid the Map Image

Note

You can use control points to warp the images and properly align the transparent image over the map image. See "Warping Image Objects" on page 121 for details.

Warping Image Objects

Object Graphics allows precise control over the color palettes used to display image objects. By initializing a palette object, both the reference image object and the transparent, warped image object can be displayed using individual color palettes.

The following example warps an African land-cover characteristics image to a political map of the continent. After displaying the images and selecting control points in each image using the XROI utility, the resulting warped image is altered to include an alpha channel, enabling transparency. Image objects are then created and displayed in an IDL Object Graphics display. Complete the following steps for a detailed description of the process.

Example Code

See `transparentwarping_object.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for code that duplicates this example. Run the example procedure by entering `transparentwarping_object` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT transparentwarping_object.pro`.

Note

For background information on warping images and selecting control points, see "Overview of Warping Images" (Chapter 5, *Image Processing in IDL*).

1. Select the political map image. This is the reference image to which the land cover image will be warped:

```
mapFile= FILEPATH('afrpolitsm.png', $
   Subdirectory = ['examples', 'data'])
```

2. Use READ_PNG routine to read in the file. Specify *mapR*, *mapG*, *mapB* to read in the image's associated color table:

```
mapImg = READ_PNG(mapFile, mapR, mapG, mapB)
```

3. Using IDLgrPalette::Init, assign the image's color table to a palette object, which will be applied to an image object in a later step:

```
mapPalette = OBJ_NEW('IDLgrPalette', mapR, mapG, mapB)
```

4. Select and open the land cover input image, which will be warped to the map:

```
landFile = FILEPATH('africavlc.png', $
   Subdirectory = ['examples', 'data'])
landImg = READ_PNG (landFile, landR, landG, landB)
```

Selecting Control Points for Image Object Warping

This section describes using the XROI utility to select corresponding control points in the two images. The arrays of control points in the input image, (*Xi, Yi*), will be mapped to the array of points selected in the reference image, (*Xo, Yo)*.

Note

The *Xi* and *Yi* vectors and the *Xo* and *Yo* vectors must be the same length, meaning that you must select the same number of control points in the reference image as you select in the input image. The control points must also be selected in the same order since the point *Xi1, Yi1* will be warped to *Xo1, Yo1*.

The following figure shows the points to be selected in the input image.

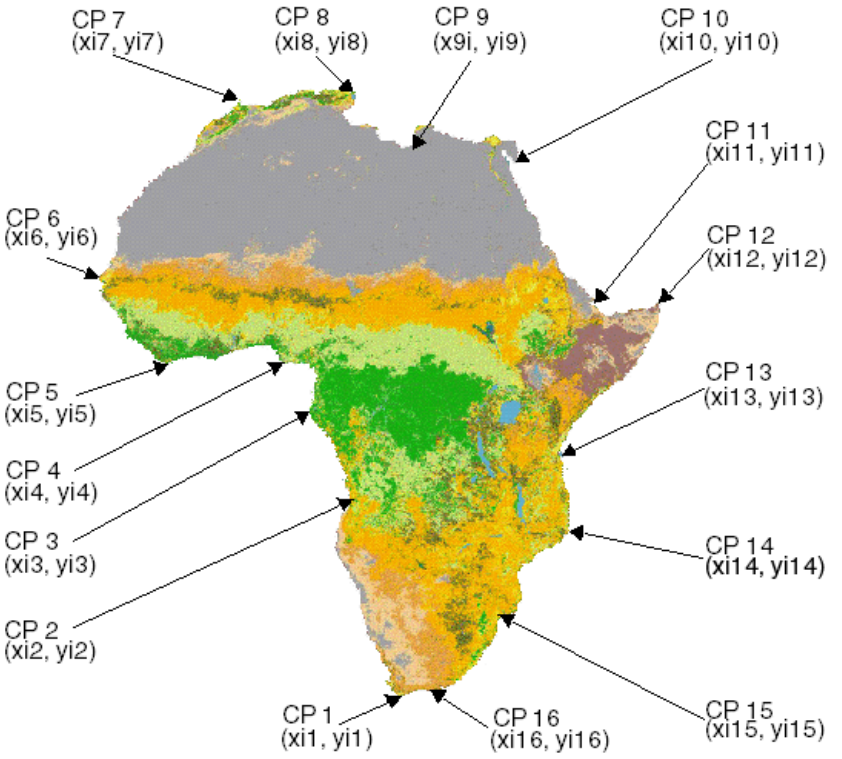


Figure 4-12: Selecting Control Points in the Input Image

Reasonably precise warping of the land classification image to the political map requires selecting numerous control points because of the irregularity of the continent's border. Select the control points in the land classification image as described in the following steps.

1. Load the image and its associated color table into the XROI utility, specifying the REGIONS_OUT keyword to save the region defined by the control points in the *landROIout* object:

```
XROI, landImg, landR, landG, landB, $
   REGIONS_OUT = landROIout, /BLOCK
```

   Select the **Draw Polygon** button from the XROI utility toolbar shown in the following figure. Position the crosshairs symbol over CP1, shown in the previous figure, and click the left mouse button. Repeat this action for each successive control point. After selecting the sixteenth control point, position the crosshairs over the first point selected and click the right mouse button to close the region. Your display should appear similar to the following figure.

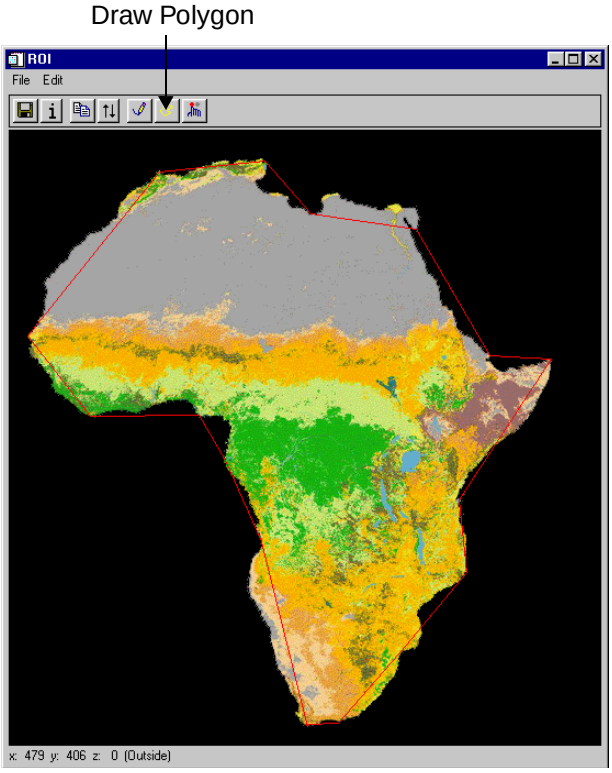


Figure 4-13: Selecting Control Points Using XROI

Note

It is of no concern that portions of the continent lie outside the polygonal boundary. The EXTRAPOLATE keyword to WARP_TRI enables warping of the image areas lying outside of the boundary of control points. However, images requiring more aggressive warp models may not have good results outside of the extent of the control points when WARP_TRI is used with the /EXTRAPOLATE keyword.

2. Close the XROI window and assign the *landROIout* object data to the *Xi* and *Yi* control point vectors:

```
landROIout -> GetProperty, DATA = landROIdata
Xi = landROIdata[0,*]
Yi = landROIdata[1,*]
```

The following figure displays the corresponding control points to be selected in the reference image of the political map. These control points will make up the *Xo* and *Yo* arrays required by the IDL warping routines.

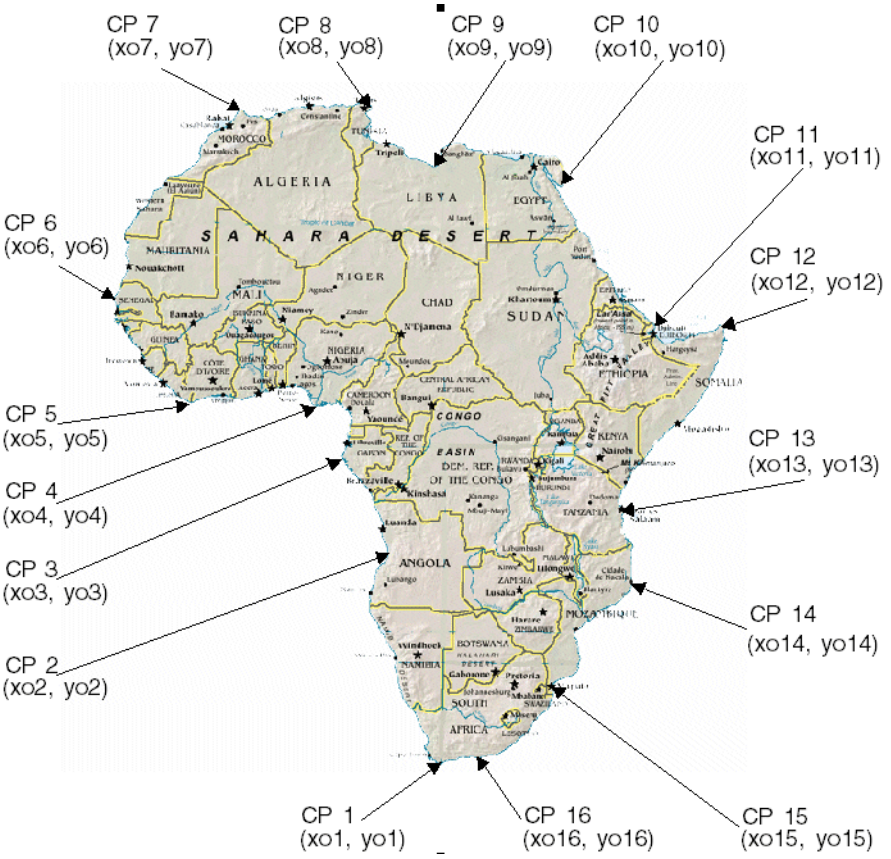


Figure 4-14: Control Points to be Selected in the Reference Image

3. Load the image of the political map and its associated color table into the XROI utility, specifying the REGIONS_OUT keyword to save the selected region in the *mapROIout* object:

```
XROI, mapImg, mapR, mapG, mapB, $
   REGIONS_OUT=mapROIout,/BLOCK
```

Select the **Draw Polygon** button from the XROI utility toolbar. Position the crosshairs symbol over CP1, shown in the previous figure, and click the left

mouse button. Repeat this action for each successive control point. After selecting the sixteenth control point, position the crosshairs over the first point selected and click the right mouse button to close the region. Your display should appear similar to the following figure.

Figure 4-15: Selecting Control Points Using XROI

4. Close the XROI window and assign the *mapROIout* object data to the *Xo* and *Yo* control point vectors:

```
mapROIout -> GetProperty, DATA=mapROIdata
Xo = mapROIdata[0,*]
Yo = mapROIdata[1,*]
```

Warping and Displaying a Transparent Image Object

The following section describes warping the land cover image to the political map and creating image objects. The resulting warped image will then be made into a transparency by creating an alpha channel for the image. Finally, the transparent object will be displayed as an overlay to the original political map.

1. Warp the input image, *landImg*, onto the reference image using WARP_TRI. This function uses the irregular grid of the reference image, defined by *Xo, Yo*, as a basis for triangulation, defining the surfaces associated with (*Xo, Yo, Xi*) and (*Xo, Yo, Yi*). Each pixel in the input image is then transferred to the appropriate position in the resulting output image as designated by interpolation. Using the WARP_TRI syntax,

   ```
   Result = WARP_TRI( Xo, Yo, Xi, Yi, Image
     [, OUTPUT_SIZE=vector][, /QUINTIC] [, /EXTRAPOLATE] )
   ```

   set the OUTPUT_SIZE equal to the reference image dimensions since this image forms the basis of the warped, output image. Use the EXTRAPOLATE keyword to display the portions of the image which fall outside of the boundary of selected control points:

   ```
   warpImg = WARP_TRI(Xo, Yo, Xi, Yi, landImg, $
      OUTPUT_SIZE=[600, 600], /EXTRAPOLATE)
   ```

2. While not required, you can quickly check the precision of the warp in a Direct Graphics display before proceeding with creating a transparency by entering the following lines:

   ```
   DEVICE, DECOMPOSED = 0
   TVLCT, landR, landG, landB
   WINDOW, 3, XSIZE = 600, YSIZE = 600, $
      TITLE = 'Image Warped with WARP_TRI'
   TV, warpImg
   ```

   Precise control point selection results in accurate warping. If there is little distortion, as in the following figure, control points were successfully selected in nearly corresponding positions in both images.

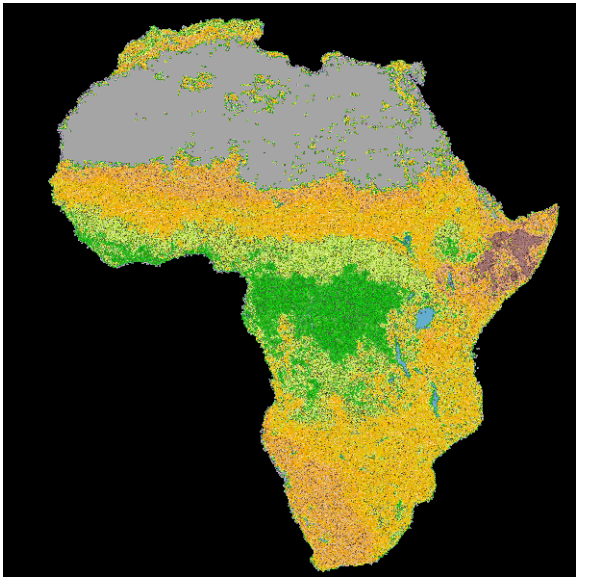


Figure 4-16: Resulting Warped Image

3. A transparent image object must be a grayscale or an RGB (24-bit) image containing an alpha channel. The alpha channel controls the transparency of the pixels. See IDLgrImage::Init for more information.

   The following lines convert the warped image and its associated color table into a RGB image containing four channels (red, green, blue, and alpha). First, get the dimensions of the warped image and then use BYTARR to create *alphaWarp*, a 4-channel by *xdim* by *ydim* array, where (*xdim*, *ydim*) are the dimensions of the warped image:

   ```
   warpImgDims = SIZE(warpImg, /Dimensions)
   alphaWarp = BYTARR(4, warpImgDims[0], warpImgDims[1],$
      /NOZERO)
   ```

4. Load the red, green and blue channels of the warped land characteristics image into the first three channels of the *alphaWarp* array:

   ```
   alphaWarp[0, *, *] = landR[warpImg]
   alphaWarp[1, *, *] = landG[warpImg]
   alphaWarp[2, *, *] = landB[warpImg]
   ```

5. Define the transparency of the alpha channel. First, create an array, masking out the black background of the warped image (where pixel values equal 0) by retaining only pixels with values greater than 0:

```
mask = (warpImg GT 0)
```

Apply the resulting mask to the alpha channel, the fourth channel of the array. This channel creates a 50% transparency of the pixels of the first three channels (red, green, blue) of the *alphaWarp* by multiplying the *mask* by 128B (byte). Alpha channel values range from 0 (completely transparent) to 255 (completely opaque):

```
alphaWarp [3, *, *] = mask*128B
```

Note

You can set the transparency of an entire image. To set the transparency of *all* pixels at 50% in this example, your could replace the two previous steps with the following two lines:

```
mask = BYTARR(s[0], s[1]) + 128
alphaWarp [3, *, *] = mask
```

6. Initialize the transparent image object using IDLgrImage::Init. Specify the BLEND_FUNCTION property of the image object to control how the alpha channel is interpreted. Setting the BLEND_FUNCTION to [3, 4] allows you to see through the foreground image to the background. The foreground opacity is defined by the alpha channel value, specified in the previous step:

```
oAlphaWarp = OBJ_NEW('IDLgrImage', alphaWarp, $
   DIMENSIONS = [600, 600], BLEND_FUNCTION = [3, 4])
```

7. Initialize the reference image object, applying the palette created earlier:

```
oMapImg = OBJ_NEW('IDLgrImage', mapImg, $
   DIMENSIONS = [600,600], PALETTE = mapPalette)
```

8. Using IDLgrWindow::Init, initialize a window object in which to display the images:

```
oWindow = OBJ_NEW('IDLgrWindow', DIMENSIONS = [600, 600], $
   RETAIN = 2, TITLE = 'Overlay of Land Cover Transparency')
```

9. Create a view object using IDLgrView::Init. The VIEWPLANE_RECT keyword controls the image display in the Object Graphics window. First create an array, *viewRect*, which specifies the *x-placement*, *y-placement*, *width*, and *height* of the view surface. The values `0, 0` place the (0, 0) coordinate of viewing surface in the lower-left corner of the Object Graphics window:

```
viewRect = [0, 0, 600, 600]
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT = viewRect)
```

10. Using IDLgrModel::Init, initialize a model object to which the images will be applied. Add the base image and the transparent alpha image to the model:

```
oModel = OBJ_NEW('IDLgrModel')
oModel -> Add, oMapImg
oModel -> Add, oAlphaWarp
```

Note

Image objects appear in the Object Graphics window in the order in which they are added to the model. If a transparent object is added to the model before an opaque object, it will not be visible.

11. Add the model, containing the images, to the view and draw the view in the window:

```
oView -> Add, oModel
oWindow -> Draw, oView
```

The following figure shows the warped image transparency overlaid onto the original reference image, the political map.

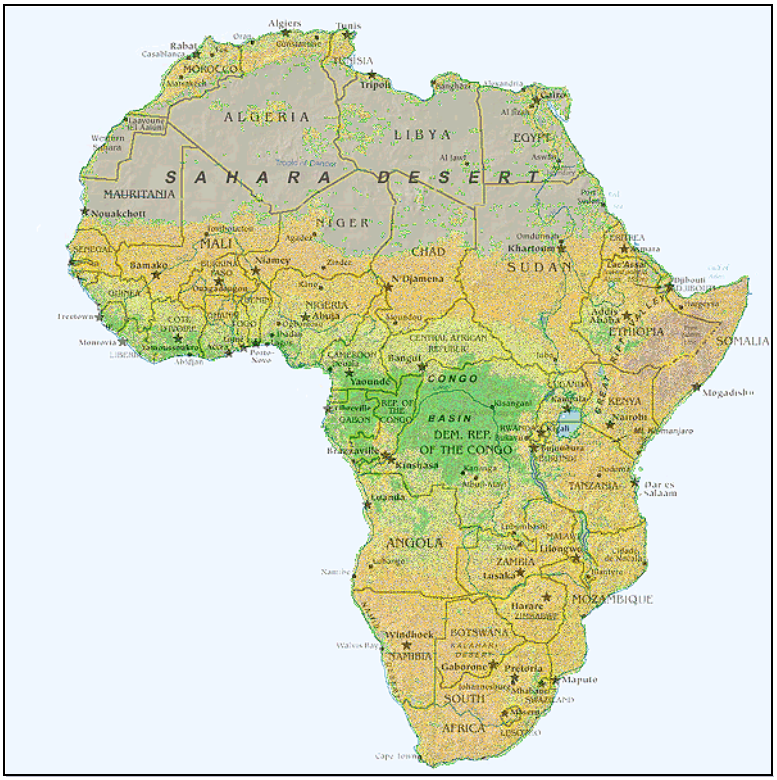


Figure 4-17: Object Graphics Display of the Political Map with a Transparent Land Cover Overlay

12. Use OBJ_DESTROY to clean up unneeded object references including the region objects:

```
OBJ_DESTROY, [oView, oMapImg, oAlphaWarp, $
   mapPalette, landROIout, mapROIout]
```

Mapping an Image Object onto a Sphere

This example maps an image containing world elevation data onto the surface of a sphere and displays the result using the XOBJVIEW utility. This utility automatically creates the window object and the view object. Therefore, this example creates an object based on IDLgrModel that contains the sphere, the image and the image palette, as shown in the conceptual representation in the following figure.

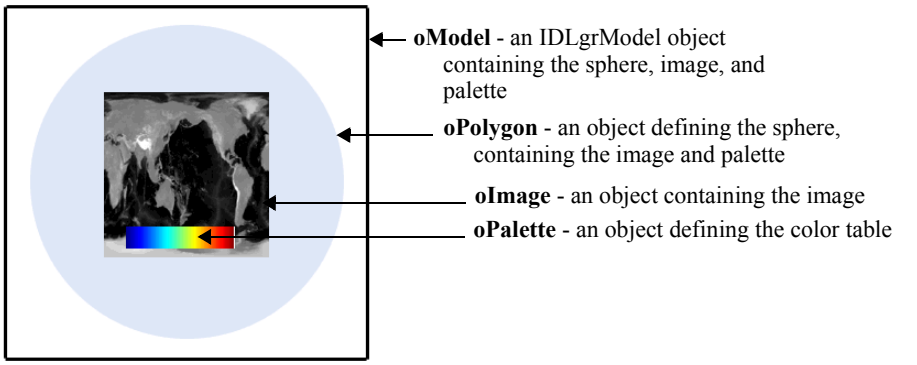


Figure 4-18: Conceptualization of XOBJVIEW Object Graphics Example

Note

For an example that maps a satellite image onto Digital Elevation Model data, see "Mapping an Image onto Elevation Data" (Chapter 3, *Image Processing in IDL*).

Complete the following steps for a detailed description of the process.

Example Code

See `maponsphere_object.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for code that duplicates this example. Run the example procedure by entering `maponsphere_object` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT maponsphere_object.pro`.

1. Select the world elevation image. Define the array, read in the data and close the file.

```
file = FILEPATH('worldelv.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

```
image = READ_BINARY(file, DATA_DIMS = [360, 360])
```

2. Use the MESH_OBJ procedure to create a sphere onto which the image will be mapped. The following invocation of MESH_OBJ uses a value of 4, which represents a spherical mesh:

   ```
   MESH_OBJ, 4, vertices, polygons, REPLICATE(0.25, 101, 101)
   ```

   When the MESH_OBJ procedure completes, the *vertices* and *polygons* variables contain the mesh vertices and polygonal mesh connectivity information, respectively. Although our image is 360 by 360, we can texture map the image to a mesh that has fewer vertices. IDL interpolates the image data across the mesh, retaining all the image detail between polygon vertices. The number of mesh vertices determines how close to perfectly round the sphere will be. Fewer vertices produce a sphere with larger facets, while more vertices make a sphere with smaller facets and more closely approximates a perfect sphere. A large number of mesh vertices will increase the time required to draw the sphere. In this example, MESH_OBJ produces a 101 by 101 array of vertices that are located in a sphere shape with a radius of 0.25.

3. Initialize the display objects. In this example, it is necessary to define a model object that will contain the sphere, the image and the color table palette. Using the syntax, *oNewObject* = OBJ_NEW('*Class_Name*'), create the model, palette and image objects:

   ```
   oModel = OBJ_NEW('IDLgrModel')
   oPalette = OBJ_NEW('IDLgrPalette')
   oPalette -> LOADCT, 33
   oPalette -> SetRGB, 255, 255, 255, 255
   oImage = OBJ_NEW('IDLgrImage', image, PALETTE = oPalette)
   ```

   The previous lines initialize the *oPalette* object with the color table and then set the final index value of the red, green and blue bands to 255 (white) in order to use white (instead of black) to designate the highest areas of elevation. The palette object is created before the image object so that the palette can be applied when initializing the image object. For more information, see IDLgrModel::Init, IDLgrPalette::Init and IDLgrImage::Init.

4. Create texture coordinates that define how the texture map is applied to the mesh. A texture coordinate is associated with each vertex in the mesh. The value of the texture coordinate at a vertex determines what part of the texture will be mapped to the mesh at that vertex. Texture coordinates run from 0.0 to 1.0 across a texture, so a texture coordinate of [0.5, 0.5] at a vertex specifies that the image pixel at the exact center of the image is mapped to the mesh at that vertex.

In this example, we want to do a simple linear mapping of the texture around the sphere, so we create a convenience vector that describes the mapping in each of the texture's *x*- and *y*-directions, and then create these texture coordinates:

```
vector = FINDGEN(101)/100.
texure_coordinates = FLTARR(2, 101, 101)
texure_coordinates[0, *, *] = vector # REPLICATE(1., 101)
texure_coordinates[1, *, *] = REPLICATE(1., 101) # vector
```

The code above copies the convenience vector through the array in each direction.

5. Enter the following line to initialize a polygon object with the image and geometry data using the IDLgrPolygon::Init function. Set `SHADING = 1` for gouraud (smoother) shading. Set the `DATA` keyword equal to the sphere defined with the MESH_OBJ function. Set `COLOR` to draw a white sphere onto which the image will be mapped. Set `TEXTURE_COORD` equal to the texture coordinates created in the previous steps. Assign the image object to the polygon object using the `TEXTURE_MAP` keyword and force bilinear interpolation:

```
oPolygons = OBJ_NEW('IDLgrPolygon', SHADING = 1, $
   DATA = vertices, POLYGONS = polygons, $
   COLOR = [255, 255, 255], $
   TEXTURE_COORD = texure_coordinates, $
   TEXTURE_MAP = oImage, /TEXTURE_INTERP)
```

Note

When mapping an image onto an IDLgrPolygon object, you must specify both TEXTURE_MAP and TEXTURE_COORD keywords.

6. Add the polygon containing the image and the palette to the model object:

```
oModel -> ADD, oPolygons
```

7. Rotate the model -90° along the *x*-axis and *y*-axis:

```
oModel -> ROTATE, [1, 0, 0], -90
oModel -> ROTATE, [0, 1, 0], -90
```

8. Display the results using XOBJVIEW, an interactive utility allowing you to rotate and resize objects:

```
XOBJVIEW, oModel, /BLOCK
```

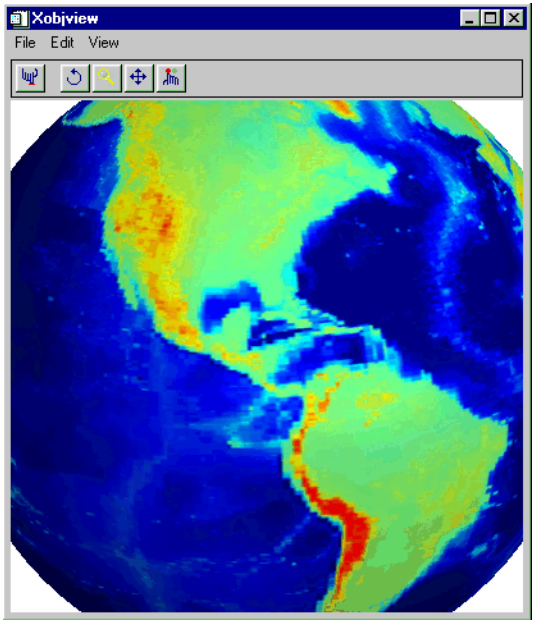


Figure 4-19: Magnified View of World Elevation Object

After displaying the object, you can rotate the sphere by clicking in the display window and dragging your mouse. Select the magnify button and click near the middle of the sphere. Drag your mouse away from the center of the display to magnify the image or toward the center of the display to shrink the image. Select the left-most button on the XOBJVIEW toolbar to reset the display. The previous figure shows a rotated and magnified view of the world elevation object.

9. After closing the XOBJVIEW display, remove unneeded object references:

```
OBJ_DESTROY, [oModel, oImage, oPalette]
```

Image Tiling

The IDLgrImage object supports tiling, which lets you display images that are too large to be read entirely into memory. For example, some satellite images can be over a gigabyte in size, which is impossible to fit into memory and display as a single unit on a typical computer. However, it can be displayed by segmenting it into smaller, more manageable image tiles.

When tiling is enabled for an IDLgrImage object, the image is initially created without any data. The image pixels are only loaded when a tile section comes into view through panning. Also, you can create an image pyramid to support level-of-detail (LOD) rendering for large images. This changes the resolution of an image when you zoom in or out within an image display. As you zoom out, successively smaller, less detailed images can be displayed. This quickly provides a full view of the larger image, lets you choose an area of interest, and zoom in on that area. As you zoom in, progressively detailed image layers can be loaded. The IDLgrImage object is aware of the LOD required and will communicate that to the application when the application requests the tile visibility information. See the following sections for more tiling information:

- “Image Pyramids” on page 137 and “Image Tiles” on page 139
- “Adding Tiling to Your Application” on page 140
- “Example: JPEG2000 Files for Tiling” on page 150

Image Pyramids

The use of image tiling and image pyramids supports the display of high-resolution images with a high level of performance. An image pyramid consists of a base image and a series of successively smaller sub-images, each at half the resolution of the previous image. The following figure shows the base image and successively smaller sub-images. The sub-images corresponds to lower resolution levels.

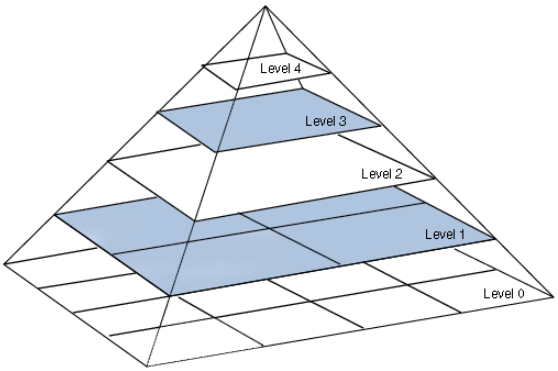


Figure 4-20: Image Pyramid

Creating Image Pyramids

You have two options if your image file does not already contain an image pyramid:

- Create an IDLffJPEG2000 object from your image data. You can define the number of levels, the size of the tiles and other properties when you create the image. The tiles and image levels are then automatically created for you.
- Create the image pyramid manually by creating a series of images, each with half the resolution of the previous image. You can use Gaussian or Laplacian filtering in combination with the subsampling if desired.

For example, taking a 4096 by 4096 base image (level 0), you could create the pyramid as follows:

Level	Resolution
0	4096 by 4096 pixels
1	2048 by 2048 pixels
2	1024 by 1024 pixels
...	...

Table 4-1: Sample Resolutions of Image Pyramid Levels

The resolution of level *n*+1 should be half that of level *n*. If level *n* is not wholly divisible by two, then level *n*+1 should be rounded down as shown in the following table.

Level and Resolution	Comment
Level 0: 20105 by 20005	Base image. Divide by 2.
Level 1: 10052 by 10002	Rounded down 20005/2 to 10002
Level 2: 5026 by 5001	Divided Level 1 by 2.
Level 3: 2513 by 2500	Rounded down 5001/2 to 2500
...	...

Table 4-2: Rounding Down Resolutions of Image Pyramid Levels

Note

See “Zooming Tiled Images” on page 143 for information on which IDLgrImage properties are typically set to take full advantage of an image pyramid, and for information on how to calculate exactly how many image levels you need based on the image and tile size.

Image Tiles

Tiling an image segments it into a number of smaller rectangular areas called tiles. If you are using a JPEG2000 image, the tile size is defined in the image, and you should use this value when creating the IDLgrImage object. If you are creating your own image pyramid, which does not have an inherent tile size defined, it is recommended that you accept the default tile size of the IDLgrImage object (1024 by 1024 pixels).

The size of the drawing area, the tile size, and the image level all play a part in the display of a tiled image. With a large, full-resolution image, only a portion of it appears in the view, so only a subset of the image tiles are displayed. In the following figure, the full-resolution, level 0 image is shown on the left. Only two of the 1024 by 1024 tiles are loaded to support what is shown in the 800 by 800 pixel drawing area, indicated by the dotted box.

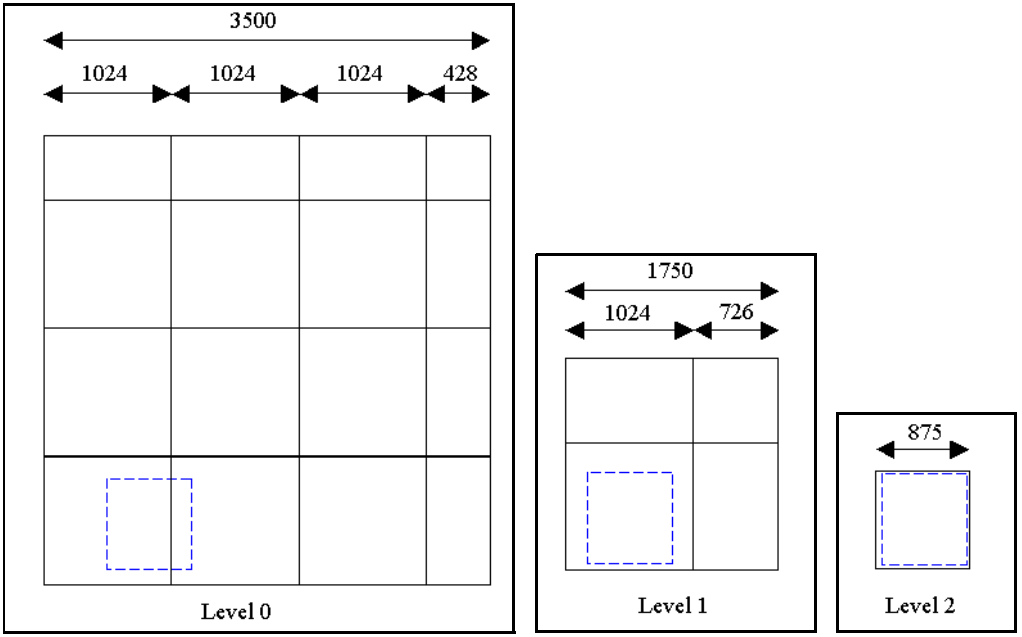


Figure 4-21: Dotted Box Showing Size of Drawing Area and Visible Tiles

If you zoom out to a zoom level of 50% or less, IDL can show the level 1 image (which is half the resolution of the level 0 image). Only a single tile is required to fill the drawing area. If you reduce the zoom level by another 50%, the level 2 image can be displayed, and the entire image is visible in the drawing area.

Adding Tiling to Your Application

Large image tiling results from the interaction between an IDLgrImage object, an IDLgrView object, and a destination object (IDLgrWindow, IDLgrClipboard, IDLgrBuffer, or IDLgrPrinter). The destination and view objects are key in determining *what* data the image object should contain. Each destination object has a QueryRequiredTiles method that determines the visible data based on the view and zoom level, and returns information about the visible image tiles. This information and image data are then passed to the image object SetTileData method. Initially, however, an image that supports tiling does not contain data.

To create an image that supports tiling, you must minimally set two IDLgrImage object properties:

- TILING = 1 enables tiling
- TILED_IMAGE_DIMENSIONS = [*width*, *height*] in pixels is the size of the image

You can also define how tiles from image levels in an image pyramid are accessed using the TILE_LEVEL_MODE mode property. Set it to 1 (automatic mode) to have IDL automatically request the proper tile level based on the zoom level. This is useful when you have an image pyramid and want to use lower resolution images when zooming out.

Note

You should not set TILE_LEVEL_MODE to automatic unless you have an image pyramid. Otherwise IDL will request non-existent lower-resolution data.

The default TILE_LEVEL_MODE value is zero (manual mode), meaning your application must specify which level should be used (where TILE_CURRENT_LEVEL defines that tile level). QueryRequiredTiles will always request tiles at this level and the image will always render using this level. This is useful if you will be panning the image without zooming. If your application does allow zooming, it is best to create an image pyramid so that you can take advantage of the memory savings afforded by displaying lower resolution images when the view is zoomed out.

Even after you have set the necessary image properties that enable tiling, the image still does not contain data. If you attempt to draw the image at this point, it will be the color of the TILE_COLOR property. You must call the QueryRequiredTiles method on the destination object (a window, printer, buffer, or clipboard object) to determine what portion of the image needs to be drawn.

Note

The following sections provide *general* information and code examples using tiling elements in IDL. For a complete, *working* example, see "Example: JPEG2000 Files for Tiling" on page 150.

Querying Required Tiles

The QueryRequiredTiles method requires references to a view object and an image object. It returns an array of structures (one for each required tile) that contains information about the tile data needed to fill the view. Once this information has been passed to the IDLgrImage object SetTileData method, call the destination object's Draw method to display the tiled image data.

For example, suppose your application displays a region of a large image (20,000 by 20,000 pixels at full resolution, where one image pixel maps to one screen pixel). Your application window is 800 by 800 pixels, which means that only this much of the image is visible at any one time. To enable tiling in this instance, create the IDLgrWindow object and then create the IDLgrImage object that supports tiling as follows:

```
oImage = OBJ_NEW('IDLgrImage', TILING=1, $
   TILED_IMAGE_DIMENSIONS=[20000,20000], $
   TILE_LEVEL_MODE=0)
```

Setting TILING=1 denotes this image will contain tile data, and TILED_IMAGE_DIMENSIONS defines the size of the full resolution image. The TILE_LEVEL_MODE=0 indicates manual level control (by default, the full resolution, level 0 image is always displayed). Not setting the TILE_DIMENSIONS accepts the default tile size, 1024 by 1024 pixels.

Initialize the IDLgrView object so the lower-left corner of the image is displayed. Where `windowDims = [800,800]`, configure the viewplane rectangle as follows:

```
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0,0,$
   windowDims[0],windowDims[1]])
```

Create a IDLgrModel object and add the image. After you add this model to the view, you can call QueryRequiredTiles to determine which tiles are visible in the view and need data as follows:

```
ReqTiles = oWindow->QueryRequiredTiles(oView, oImage, $
   COUNT=nTiles)
```

`ReqTiles` is an `nTiles` element array of named structures describing the tiles required. See "IDLgrWindow::QueryRequiredTiles" (*IDL Reference Guide*) for information on the fields in this structure. The destination objects that support tiling

share this method and named structure. Your application will need to iterate through this array, extracting the tile data from the image data and passing it to IDLgrImage::SetTileData.

For a TIFF image (`largeimage.tif`), you can use the READ_TIFF routine's SUB_RECT keyword to extract the tile data from the image as follows:

```
FOR i = 0, nTiles - 1 DO BEGIN
   SubRect = [ReqTiles[i].X, ReqTiles[i].Y, $
      ReqTiles[i].Width, ReqTiles[i].Height]
   TileData = READ_TIFF('largeimage.tif', SUB_RECT=SubRect)
   TileData = BYTSCL(TileData, MIN=0, MAX=1024)
   oImage->SetTileData, ReqTiles[i], TileData
ENDFOR
```

For a JPEG2000 image (`oJP2File`) with the same `SubRect` variable as that defined in the previous example, you can use the IDLffJPEG2000::GetData method's REGION keyword to extract the data as follows:

```
; Load the data.
TileData = oJP2File->GetData(REGION=SubRect)
oImage->SetTileData, ReqTiles[i], TileData
```

When the destination object's Draw method is called, the display will contain the correct portion of the image since the data associated with the visible tiles has been loaded.

Note

You do not need to pass only a single tile to SetTileData. You can pass a row of tiles or load tiles without a prior call to QueryRequiredTiles (tile caching). See "Preloading Tiles" on page 147 for details.

Panning Tiled Images

To pan an image, you can query and assign the tile data without regard for image level as shown in the previous section, "Querying Required Tiles" on page 141. Panning is accomplished by changing the *x* and *y* elements of the view object's VIEWPLANE_RECT property, where [*x*, *y*, *width*, *height*] describe the visible view area. After changing the VIEWPLANE_RECT, call the window object's QueryRequiredTiles method to determine if new data is required. If so, load the data as before. The following code shows an example of modifying the viewplane for panning:

```
oView->GetProperty, VIEWPLANE_RECT=vp

; Panning. This is done by changing the position of the
```

```
; VIEWPLANE_RECT (vp) which is described by [x,y,width,height]
; where x and y are the lower-left corner. How far to move it
; is computed from the distance of the mouse from the center of
; the window (xDelta,yDelta) and the 'zoom factor',
; (vp[2] / windowDims[0]), which is the viewplane
; width divided by the window x dimension. The farther
; the cursor is from the center of the window, the faster the
; view pans.
factor = (vp[2] / windowDims[0])
vp[0] += xDelta * factor
vp[1] += yDelta * factor
(*pState).oView->SetProperty, VIEWPLANE_RECT=vp
```

See “Example: JPEG2000 Files for Tiling” on page 150 for information on where to locate the full tiling example.

Zooming Tiled Images

As with panning, you can use the view object’s VIEWPLANE_RECT to zoom (by changing the *width* and *height* elements). However, care must be taken when zooming out as many tiles of high-resolution data may need to be loaded, which could exhaust the tile cache. It is best to enable zooming for very large images only when you have an image pyramid of lower resolution images.

When you zoom out to view more of an image, multiple image pixels are mapped to a single pixel on the screen. When dealing with large tiled images, you can take advantage of this situation by displaying a series of lower resolution images (an image pyramid), which uses memory more efficiently. There is no need to use the full resolution image. For example, say you have an image that is 20,000 by 20,000 pixels and over 300 MB. Without an image pyramid, if you zoom out so that the entire image is visible in an 800 by 800 pixel view, the entire image (381 MB) will be loaded into memory. While this might be possible, it isn’t efficient. With an image pyramid, you could easily display a lower resolution image that fit the window size. This image would likely be less than one MB in size, would easily fit into memory, and would still be of a sufficient resolution for identifying general areas of interest.

Note

Unless the image file format automatically includes an image pyramid (such as JPEG2000 files), you will need to either create a JPEG2000 file that contains your image data or create an image pyramid manually. See “Image Pyramids” on page 137 for details.

The following code shows how to modify the viewplane rectangle associated with the view object to support zooming:

```
; Zooming. This is done by changing the position and dimensions
; of the VIEWPLANE_RECT (vp), is described by [x,y,width,height]
; where x and y are the lower-left corner. When zooming in, a
; smaller portion of the total image is displayed in the viewplane
; rectangle, which is reflected in smaller vp width and height
; values. The rectangle size is computed from:
;    factor -  the vp width divided by the window x dimension.
;    delta  -  yDelta (the absolute vertical change from the
             ; center of the image times the factor. The
             ; further the mouse cursor is from the center,
             ; the faster the zoom.
;    aspect -  the window y dimension divided by the x dimension.

factor = (vp[2] / windowDims[0])
delta  = yDelta * factor
aspect = float(windowDims[1]) / windowDims[0]
vp[0] += delta/2
vp[1] += delta * aspect /2
vp[2] -= delta
vp[3] -= delta * aspect
oView->SetProperty, VIEWPLANE_RECT=vp
zoom = windowDims[0] / vp[2]
```

See “Example: JPEG2000 Files for Tiling” on page 150 for information on where to locate the full tiling example. See “Using Image LEVEL When Zooming” on page 145 for information on how to request tile data based on zoom level.

Calculating the Number of Image Pyramid Levels

When you have an image pyramid, you will want to set the IDLgrImage TILE_LEVEL_MODE property to 1 (automatic). Doing so causes TILE_NUM_LEVELS to automatically calculate the number of levels needed unless you set a different value. IDL requests levels up to TILE_NUM_LEVELS - 1. This property is based on the original (level 0) image size and the tile size such that the lowest resolution image is just slightly smaller than the tile size. See “TILE_NUM_LEVELS” (*IDL Reference Guide*) for an example.

To figure out how many levels are needed, create an image object with dimensions equal to the dimensions of your image, the tile mode to automatic, and tiling equal to 1. For example, for the 20,000 by 20,000 image, with the default tile size (1024 by 1024 pixels), create an image object as follows:

```
oImage = OBJ_NEW('IDLgrImage', $
   TILED_IMAGE_DIMENSIONS=[20000,20000], $
   TILING=1, TILE_LEVEL_MODE=1)
```

Here TILE_LEVEL_MODE is set to `1` (automatic) so the level requested by the destination object's QueryRequiredTiles method is calculated automatically from the view information. Return the number of levels that are needed in an image pyramid as follows:

```
oImage->GetProperty, TILE_NUM_LEVELS=nLevels
```

The `nLevels` variable contains the number of levels IDL will request. You will need `nLevels - 1` levels in your pyramid since level 0 is the full resolution image.

Using Image LEVEL When Zooming

In an application that has an image pyramid and supports zooming, you will use information returned by QueryRequiredTiles to load different resolution image tiles. As in the basic query example ("Querying Required Tiles" on page 141), create and initialize the view so the lower-left corner of the image is initially displayed:

```
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0, 0, 800, 800])
```

Again, create an IDLgrModel, and add the image. Once the model has been added to the view (not shown), call QueryRequiredTiles to determine which tiles are visible and need data.

```
ReqTiles = oWindow->QueryRequiredTiles(oView, oImage, $
   COUNT=nTiles)
```

Rather than reading from the original, full-resolution image, determine which image to use based on the LEVEL field of the returned structure contained in `ReqTiles`. If you have created an image pyramid for TIFF images, consider using the following naming scheme to return the correct resolution image based on the LEVEL field:

```
filenames = strarr(6)
filenames[0] = 'largeimage.tif'  ; Full-resolution image
filenames[1] = 'largeimage1.tif' ; Half-resolution image
filenames[2] = 'largeimage2.tif' ; Quarter-resolution image
filenames[3] = 'largeimage3.tif' ; Eighth-resolution image
filenames[4] = 'largeimage4.tif' ; 1/16-resolution image
filenames[5] = 'largeimage5.tif' ; 1/32-resolution image
```

You can then request the correct image level (`level`) and set tile data as follows:

```
FOR i = 0, nTiles - 1 DO BEGIN
   SubRect = [ReqTiles[i].X, ReqTiles[i].Y, $
      ReqTiles[i].Width, ReqTiles[i].Height]
   level = ReqTiles[i].Level
   TileData = READ_TIFF(filenames[Level], SUB_RECT=SubRect)
   TileData = BYTSCL(TileData, MIN=0, MAX=1024)
   oImage->SetTileData, ReqTiles[i], TileData
ENDFOR
```

For a JPEG2000 image (`oJP2File`), you can use the IDLffJPEG2000::GetData method's DISCARD_LEVELS keyword to return the correct image level as follows:

```
FOR i = 0, nTiles - 1 DO BEGIN
   SubRect = [ReqTiles[i].x, ReqTiles[i].y, $
      ReqTiles[i].width, ReqTiles[i].height]

   ; Convert to JPEG2000 canvas coords.
   level = ReqTiles[i].level
   Scale = ISHFT(1, level)
   SubRect = SubRect * Scale

   ; Load the data.
   TileData = oJP2File->GetData(REGION=SubRect, $
      DISCARD_LEVELS=level, ORDER=1)
   oImage->SetTileData, ReqTiles[i], TileData
ENDFOR
```

An image that supports TILE_LEVEL_MODE=1 (automatic) can be panned and zoomed using VIEWPLANE_RECT and the above QueryRequiredTiles and SetTileData combination. This determines tile visibility and loads the appropriate data. As the image is zoomed out, lower resolution data will be automatically requested to ensure physical memory does not run out.

Note

See "Example: JPEG2000 Files for Tiling" on page 150 for the complete JPEG2000 tiling example.

Copying and Printing a Tiled Image

The IDLgrClipboard and IDLgrPrinter objects have a QueryRequiredTiles method just like IDLgrWindow. Return the visible tiles using QueryRequiredTiles, set the data on the image object, and use the Draw method of the printer or clipboard object to output the portion of the tiles that are visible in the view. This is all that is required for a clipboard object. For a printer object, you need to take the view dimensions into account when printing. The following code excerpt shows this for the object, `oPrinter`:

```
; Set the dimensions of the view so the aspect ratio is
; correct when printed.
windowAspect = FLOAT(windowDims[0]) / windowDims[1])
oPrinter->GetProperty, DIMENSIONS = pageSize
pageSize[1] = pageSize[0] / windowAspect
oView->SetProperty, DIMENSIONS=pageSize
```

Call QueryRequiredTiles on the printer object and set the tile data using SetTileData on the image object (as described in “Querying Required Tiles” on page 141). It is then simple to print the output:

```
;...PRINT!...
oPrinter->Draw, oView, VECTOR=0
oPrinter->NewDocument
```

Note

Clipboard and printer vector output (VECTOR=1) is not supported for tiled images.

An example in the IDL distribution provides working examples of copying and printing a tiled image. See “Example: JPEG2000 Files for Tiling” on page 150 for information on where to locate the full tiling example.

Preloading Tiles

You can load more tiles of data than what are currently visible in a view in a couple of ways:

- Pass a row of tile data to SetTileData based on an initial query (see “Loading a Row of Tiles” on page 147 below)
- Pass data to SetTileData without a query (see “Caching Non-Visible Tiles” on page 148)

Loading a Row of Tiles

SetTileData can accept more than a single tile's worth of data in one call. In some cases, it can be more efficient to read an entire row of tiles rather than extract single tiles from that row. In general, raw binary image formats (such as TIFF) that are not stored on disk in a blocked manner can be tiled more efficiently by passing rows of data. The following code shows how to load entire scanlines at once when using these image formats.

As in the example shown in “Querying Required Tiles” on page 141, which creates view and image objects, call the destination object’s QueryRequiredTiles method to determine what tile data is initially visible in the viewport as follows:

```
; Return structure information for visible tiles.
ReqTiles = oWindow->QueryRequiredTiles(oView, oImage, $
   COUNT=nTiles)

; While there are tiles, determine the width of the information to
; be requested by dividing the width of the original image by the
; current level.
WHILE (nTiles gt 0) DO BEGIN
```

```
    TileInfo = ReqTiles[0]
    level = TileInfo.Level
    width = imageDims[0] / (2 ^ level)

    ; Set the area to be read (equal to image width) to the SubRect
    ; variable.
    SubRect = [0, TileInfo.Y, width, TileInfo.Height]

    ; Insert code here to read the tile data, passing SubRect to the
    ; correct data access procedure for your file type.
    ; ...

    ; Update the tile structure.
    TileInfo.X = 0L
    TileInfo.Width = width

    ; Set the row of tile data to the image.
    oImage->SetTileData, TileInfo, TileData

    ReqTiles = oWindow->QueryRequiredTiles(oView, oImage, $
       COUNT=nTiles)
ENDWHILE
```

`SubRect` is set such that the entire width of the image is read at the requested level for the given vertical position and height. The tile structure is updated to reflect the fact that the information being passed to SetTileData starts at `X=0` and is the entire width of the image. Notice that rather than iterate through the entire `ReqTiles` array the code calls QueryRequiredTiles again after calling SetTileData since the remaining tiles in the array can now be loaded.

Caching Non-Visible Tiles

You do not need to call QueryRequiredTiles before passing data to the SetTileData method. The QueryRequiredTiles call just limits the requested data to those tiles that are visible in the view. Setting tile data without first requesting it has a couple of important uses: loading an entire level and predictive tile caching.

When an application starts, it can automatically load an entire level of low-resolution tile data. If higher resolution data is requested but not currently available, the lower resolution tiles are used. For example, if level 3 tile data has been loaded, but you attempt to zoom in so that the level 0 data is needed, level 3 data will continue to be displayed until the higher resolution data can be loaded. This results in a blurred or blocky version of the image, which can still be used until the required level has been loaded.

To load an entire level (assumed to be level 3, 2500 by 2500 pixels in this example), you first need to request that level of data from your image pyramid. How you access

this data depends on the file type. For example, if you have created a series of TIFF files, access the image data using the file name:

```
TileData = READ_TIFF('largeimage3.tif')
```

If you have created a JPEG2000 image, access the image data using the GetData method where level should be set to the level of data you want to return (e.g., 3):

```
TileData = oJP2File->GetData(DISCARD_LEVELS=level)
```

Create a tile structure that encompasses the entire level and pass the data to SetTileData:

```
tile = { IDLIMAGETILE, X:0, Y:0, Width:2500, Height:2500, $
   Level:3, Dest:oWindow }
oImage->SetTileData, tile, TileData
```

The second use for preloading tiles is predictive tile loading. For example, if the user is panning right, but tiles to the right of the view that are not yet visible, these tiles can be preloaded if there is any idle time. Then when the view reaches those tiles, there will be no interruption as the tiles have already been loaded.

Example: JPEG2000 Files for Tiling

The tiling example provided in the IDL distribution takes a 5000 by 5000 pixel JPEG file containing an aerial photograph of Chicago's O'Hare International Airport and creates a JPEG2000 file from the data. This file type provides inherent support for image tiles.

Example Code

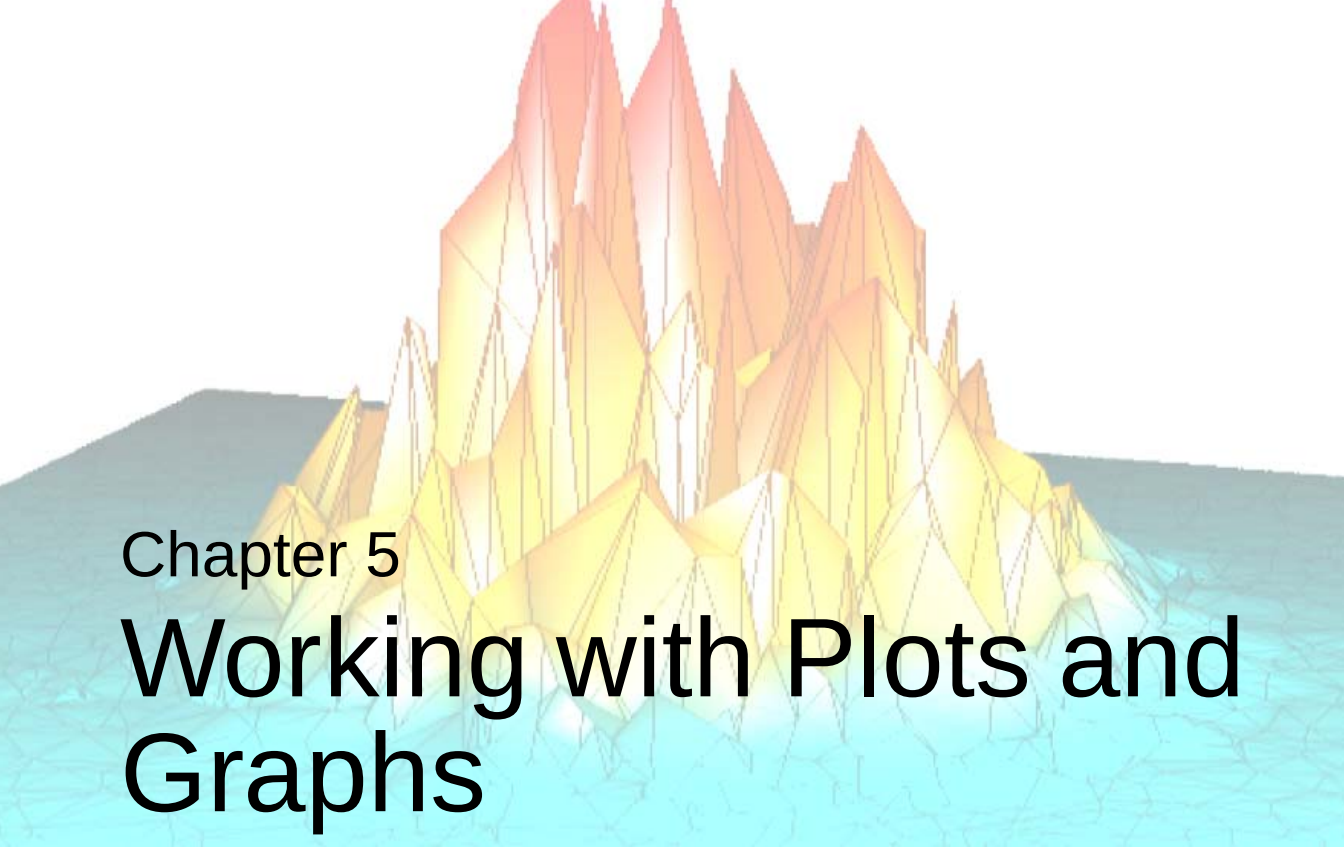
See `tilingjp2_doc.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for the tiling application code. Run the example procedure by entering `tilingjp2_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT tilingjp2_doc.pro`.

Note

The first time you run this application, it generates the JPEG2000 file. This might take a noticeable amount of time, depending on your system speed. However, once the JPEG2000 image is created, this file will be used instead of being recreated.

The following figure shows the O'Hare image. When the application opens, the view is positioned in the upper-left corner of the full-resolution image.

Figure 4-22: O'Hare Image



Chapter 5
Working with Plots and Graphs

This chapter describes the use of contour, polygon, polyline, and plot objects to create plots and graphs. The following topics are covered in this chapter:

Contour Objects

Contour objects create a set of contour lines from data stored in a rectangular array or in a set of unstructured points. Contour objects can consist either of lines or of filled regions.

Creating Contour Objects

To create a contour object, provide a vector or two-dimensional array containing the values to be contoured to the IDLgrContour::Init method. For example, the following statement creates a contour from a two-dimensional array returned by the IDL DIST function:

```
mycontour = OBJ_NEW('IDLgrContour', DIST(20))
```

See "IDLgrContour" (*IDL Reference Guide*) for details on creating contour objects.

Using Contour Objects

Contour objects have a number of properties that determine how they are rendered. See "IDLgrContour Properties" (*IDL Reference Guide*) for a complete listing. The following code displays the contour object created above in the *X*-*Y* plane.

Note

In order to display the contour as on the plane (rather than as a three-dimensional image), you must set the PLANAR property of the contour object equal to one and explicitly set the GEOMZ property equal to zero.

```
mywindow = OBJ_NEW('IDLgrWindow')
myview = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0,0,19,19])
mymodel = OBJ_NEW('IDLgrModel')
data = DIST(20)
mycontour = OBJ_NEW('IDLgrContour', data, COLOR=[100,150,200], $
   C_LINESTYLE=[0,2,4], /PLANAR, GEOMZ=0, C_VALUE=INDGEN(20))

myview->Add, mymodel
mymodel->Add, mycontour
mywindow->Draw, myview
```

This results in the following figure.

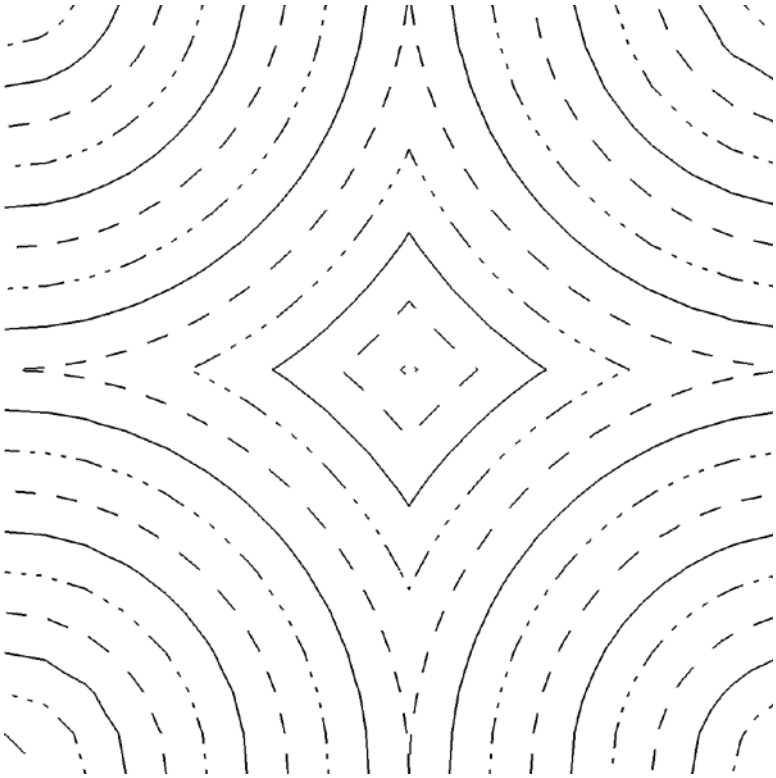


Figure 5-1: Contour Object

A more complex example using a contour object is shown in the contour demo. To start the demos, type `demo` at the IDL command prompt. Both the terrain elevation and vehicle tire data sets are displayed using the contour object.

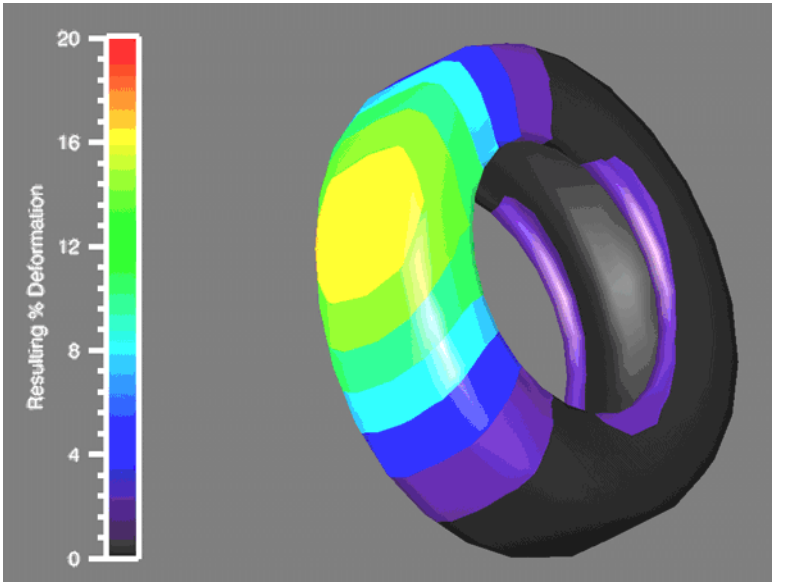


Figure 5-2: Complex Contour Object

Plot Objects

Plot objects maps a set of abscissa values to a set of ordinate values and creates a polyline connecting the points. Note that plot objects do not automatically create axes for the plot lines they create.

Creating Plot Objects

Create a plot line by providing a vector of *Y* values, and, optionally, a vector of *X* values. If no *X* values are provided, the *Y* values are plotted against the element indices of the *Y* vector.

The following statement creates a plot object plotting the values [2, 9, 4, 4, 6, 2, 8] against their own indices:

```
myplot = OBJ_NEW('IDLgrPlot', [2,9,4,4,6,2,8])
```

The following statements plot the same data versus a series of primes:

```
datay = [2,9,4,4,6,2,8]
datax = [0,1,2,5,7,11,13]
myplot = OBJ_NEW('IDLgrPlot', datax, datay)
```

See “IDLgrPlot” (*IDL Reference Guide*) for details on creating plot objects.

Using Plot Objects

Plot objects can be configured to draw regular *X* vs. *Y*, histogram, or polar plots. Set the HISTOGRAM property to create a histogram plot, or the POLAR property to create a polar plot. The following example uses the same data set to create a standard plot, a histogram plot, and a standard plot using a boxcar filter. All three plots are displayed in the same view.

```
mywindow = OBJ_NEW('IDLgrWindow')
myview = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[-10,-10,20,20])
mymodel = OBJ_NEW('IDLgrModel')

x = (FINDGEN(21) / 10.0 - 1.0) * 10.0
y = [3.0, -2.0, 0.5, 4.5, 3.0, 9.5, 9.0, 4.0, 1.0, -8.0, $
    -6.5, -7.0, -2.0, 5.0, -1.0, -2.0, -6.0, 3.0, 5.5, 2.5, -3.0]
myplot1 = OBJ_NEW('IDLgrPlot', x, y, COLOR=[120, 120, 120])
myplot2 = OBJ_NEW('IDLgrPlot', x, y, /HISTOGRAM, LINESTYLE=4)
y2 = SMOOTH(y, 5)
myplot3 = OBJ_NEW('IDLgrPlot', x, y2, LINESTYLE=2)

myview->Add, mymodel
```

```
mymodel->Add, myplot1
mymodel->Add, myplot2
mymodel->Add, myplot3
mywindow->Draw, myview
```

Figure 5-3: Plot Object

Minimum and Maximum Values

You can control the minimum and maximum values of data plotted by a plot object. Set the MAX_VALUE property of the plot object to disregard data values higher than a specified value. Set the MIN_VALUE property to disregard data values lower than a specified value. Floating-point Not-a-Number (NaN) values are also treated as missing data and are not plotted.

For example, the following statement changes the minimum and maximum values of the histogram plot, and re-draws the view object:

```
myplot2->SetProperty, MAX_VALUE=8, MIN_VALUE=2
mywindow->Draw, myview
```

Using Plotting Symbols

Set the SYMBOL property of a plot object equal to the object reference of a symbol object to display that symbol at each data point. For example, to use a triangle symbol at each data point, create the following symbol object, set the plot object's SYMBOL property, and re-draw:

```
mySymbol = OBJ_NEW('IDLgrSymbol', 5, SIZE=[.3,.3])
myplot1->SetProperty, SYMBOL=mySymbol
mywindow->Draw, myview
```

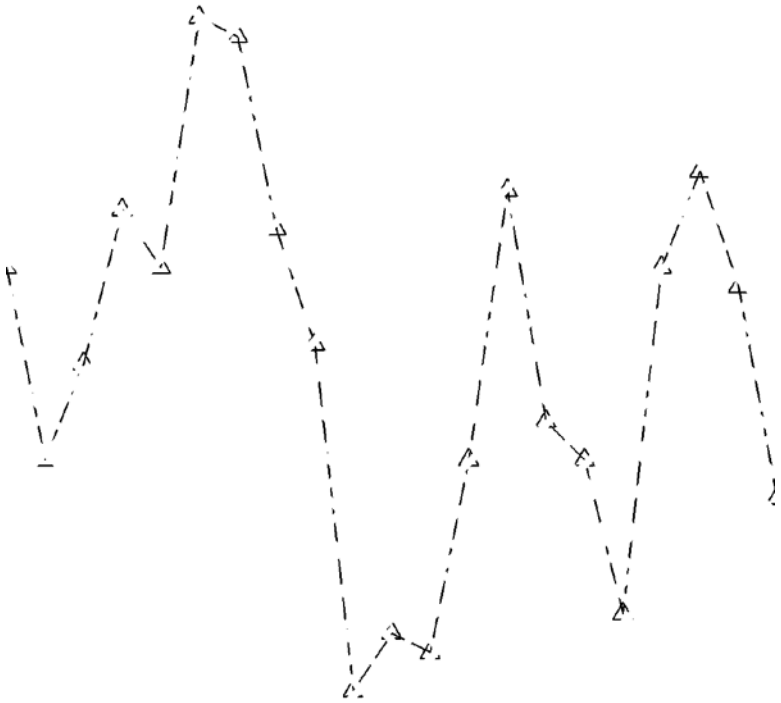


Figure 5-4: Plotting Symbols

Averaging Points

Use the NSUM property of the plot object to average the values of a group of data points before plotting. If there are m data points, m/NSUM data points are plotted. For example, the following statement causes IDL to average pairs of data points when plotting the line for the histogram plot.

```
myplot2->SetProperty, NSUM=2
mywindow->Draw, myview
```

Polar Plots

To create a polar plot, provide a vector of radius values, a vector of theta values, and set the POLAR property to a nonzero value. The following example creates a simple polar plot:

```
mywindow = OBJ_NEW('IDLgrWindow')
myview = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[-100,-100,200,200])
mymodel = OBJ_NEW('IDLgrModel')
r = FINDGEN(100)
theta = r/5
mypolarplot = OBJ_NEW('IDLgrPlot', r, theta, /POLAR)
myview->Add, mymodel
mymodel->Add, mypolarplot
mywindow->Draw, myview
```

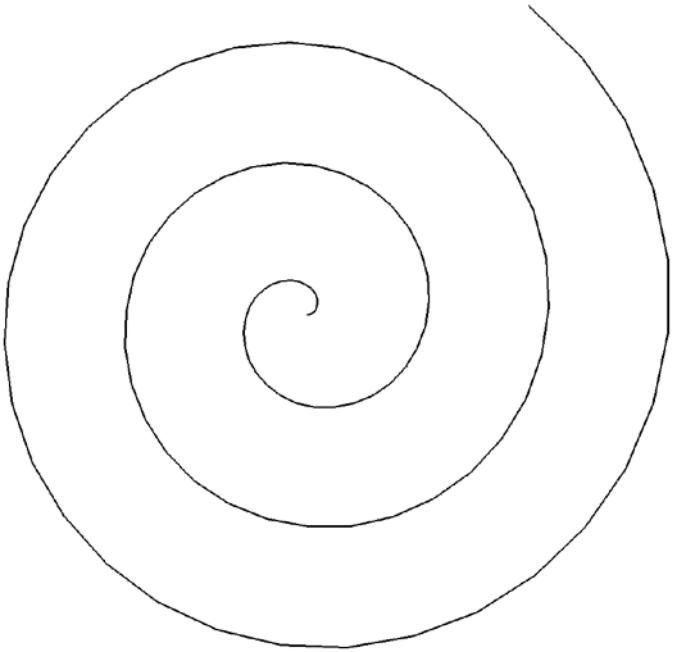


Figure 5-5: Polar Plot

Axis Objects

Axis objects provide a visual notation of data values in two- and three-dimensional plots and graphs. Each axis is represented by an individual axis object; that is, if you have a plot in *X* and *Y*, you will need to create an *x*-axis object and a *y*-axis object.

Note

Axis objects do not take their range values from data values or other objects, as you might expect if you are familiar with IDL Direct Graphics. Instead, axis objects have a default range of 0.0 to 1.0; you must explicitly set the range of values covered by the axis object using the RANGE property.

Creating Axis Objects

To create an axis object, specify an integer argument to the IDLgrAxis::Init method when calling OBJ_NEW. Specify 0 (zero) to create an *x*-axis object, 1 (one) to create a *y*-axis object, or 2 to create a *z*-axis object:

```
xaxis = OBJ_NEW('IDLgrAxis', 0)
yaxis = OBJ_NEW('IDLgrAxis', 1)
zaxis = OBJ_NEW('IDLgrAxis', 2)
```

The various keywords to the Init method allow you to control the number of major and minor ticks, the tick length and direction, the data range, and other attributes. For example, to create an *x*-axis object whose data range is between –5 and 5, with the tick marks below the axis line, use the following command:

```
xaxis = OBJ_NEW('IDLgrAxis', 0, RANGE=[-5.0, 5.0], TICKDIR=1)
```

To suppress minor tick marks:

```
xaxis->SetProperty, MINOR=0
```

See “IDLgrAxis” (*IDL Reference Guide*) for details on creating axis objects.

Using Axis Objects

Suppose you wish to create an *X*-*Y* plot of some data and wish to include both *x*- and *y*-axes.

Example Code

The following example code is included in a procedure file named `obj_axis.pro`, located in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `obj_axis` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT obj_axis.pro`.

First, we create some data to plot, the plot object, and the axis objects:

```
data = FINDGEN(100)
myplot = OBJ_NEW('IDLgrPlot', data)
xaxis = OBJ_NEW('IDLgrAxis', 0)
yaxis = OBJ_NEW('IDLgrAxis', 1)
```

Next, we retrieve the data range from the plot object and set the *x*- and *y*-axis objects' RANGE properly so that the axes will match the data when displayed:

```
myplot->GetProperty, XRANGE=xr, YRANGE=yr
xaxis->SetProperty, RANGE=xr
yaxis->SetProperty, RANGE=yr
```

By default, major tickmarks are 0.2 data units in length. Since the data range in this example is 0 to 99, we set the tick length to 2% of the data range instead:

```
xtl = 0.02 * (xr[1] - xr[0])
ytl = 0.02 * (yr[1] - yr[0])
xaxis->SetProperty, TICKLEN=xtl
yaxis->SetProperty, TICKLEN=ytl
```

Create model and view objects to contain the object tree, and a window object to display it:

```
mymodel = OBJ_NEW('IDLgrModel')
myview = OBJ_NEW('IDLgrView')
mywindow = OBJ_NEW('IDLgrWindow')
mymodel->Add, myplot
mymodel->Add, xaxis
mymodel->Add, yaxis
myview->Add, mymodel
```

Use the SET_VIEW procedure to add an appropriate viewplane rectangle to the view object. (See "Finding an Appropriate View Volume" on page 78 for information on SET_VIEW).

```
SET_VIEW, myview, mywindow
```

Now, display the plot:

```
mywindow->Draw, myview
```

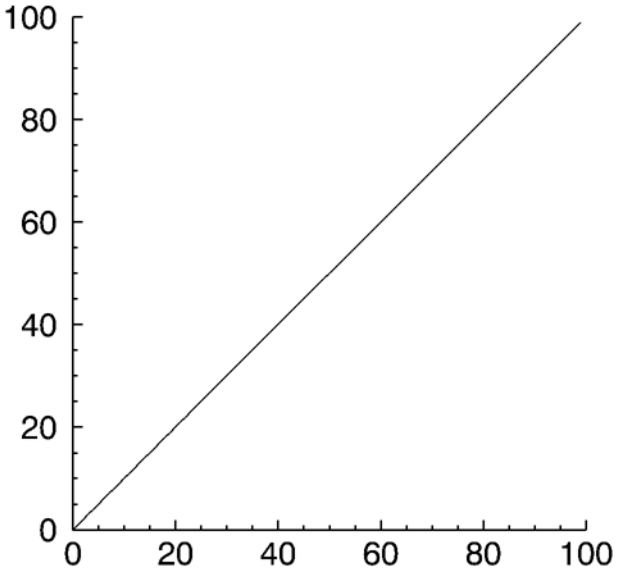


Figure 5-6: Axis Object

Logarithmic Axes

Creating a plot of logarithmic data requires that you create a logarithmic axis as well. The example referenced here first creates a linear plot, then takes a logarithm of the same data and creates a log-linear plot.

Example Code

The example code for logarithmic axes is included in a procedure file named `obj_logaxis.pro`, located in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `obj_logaxis` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT obj_logaxis.pro`.

When you run this example, notice that you need to position your mouse cursor at the IDL command prompt and hit you Enter key to step through the program and arrive at the following output.

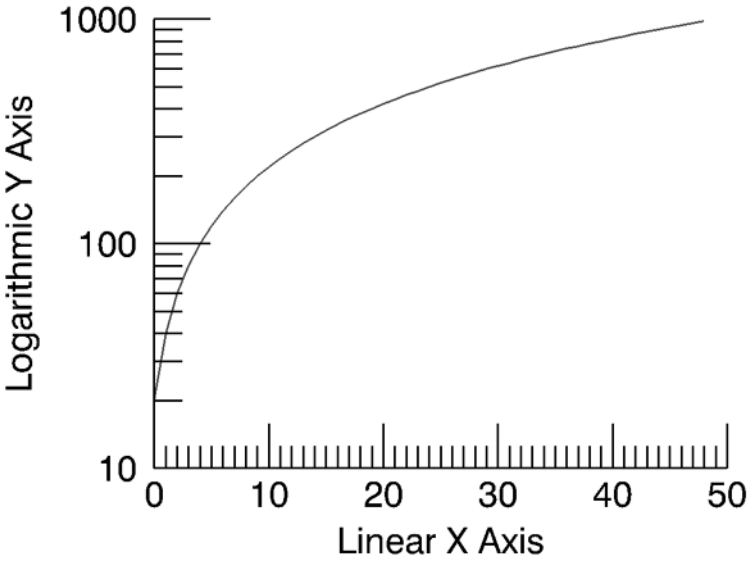


Figure 5-7: Logarithmic Axes

Displaying Date/Time Data on Axis Objects

Dates and times are among the many types of information that numerical data can represent. IDL provides a number of routines that offer specialized support for generating, analyzing, and displaying date- and time- based data (herein referred to as date/time data). For information on Julian dates and times, the Precision of Date/Time data, and information on how to generate Date/Time data, see “Date/Time Data” (Chapter 13, *Application Programming*).

You can display date/time data on plots, contours, and surfaces through the tick settings of the date/time axis. Date/time data can be displayed on any axis (x, y or z). The date/time data is stored as Julian dates, but the LABEL_DATE routine and axis keywords allow you to display this data as calendar dates. The following examples show how to display one-dimensional and two-dimensional date/time data:

- “Displaying Date/Time Data on a Plot Display” below
- “Displaying Date/Time Data on a Contour Display” on page 170

Displaying Date/Time Data on a Plot Display

Date/time data usually comes from measuring data values at specific times. For example, the displacement (in inches) of an object might be recorded at every second for 37 seconds after the initial recording of 59 minutes and 30 seconds after 2 o'clock pm (14 hundred hours) on the 30th day of March in the year 2000 as follows

```
number_samples = 37
date_time = TIMEGEN(number_samples, UNITS = 'Seconds', $
   START = JULDAY(3, 30, 2000, 14, 59, 30))
displacement = SIN(10.*!DTOR*FINDGEN(number_samples))
```

Normally, this type of data would be imported into IDL from a data file. However, this section is designed specifically to show how to display date/time data, not how to import data from a file; therefore, the data for this example is created with the above IDL commands.

Before displaying this one-dimensional data with the IDLgrPlot object, the format of the date/time values is specified through the LABEL_DATE routine:

```
date_label = LABEL_DATE(DATE_FORMAT = ['%I:%S'])
```

where %I represents minutes and %S represents seconds.

Before applying the results from LABEL_DATE, we must first create (initialize) our display objects:

```
oPlotWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [800, 600])
oPlotView = OBJ_NEW('IDLgrView', /DOUBLE)
oPlotModel = OBJ_NEW('IDLgrModel')
oPlot = OBJ_NEW('IDLgrPlot', date_time, displacement, $
   /DOUBLE)
```

The oPlotModel object will contain the IDLgrPlot and IDLgrAxis objects. The oPlotView object contains the oPlotModel object with the DOUBLE keyword. The DOUBLE keyword is set for the oPlotView and oPlot objects because the date/time data is made up of double-precision floating-point values.

Although the date/time part of the data will actually be contained and displayed through the IDLgrAxis object, the oPlot object is created first to provide a display region for the axes:

```
oPlot->GetProperty, XRANGE = xr, YRANGE = yr
xs = NORM_COORD(xr)
xs[0] = xs[0] - 0.5
ys = NORM_COORD(yr)
ys[0] = ys[0] - 0.5
oPlot->SetProperty, XCOORD_CONV = xs, YCOORD_CONV = ys
```

The NORM_COORD routine is used to create a normalized (0 to 1) display coordinate system. This coordinate system will also apply to the IDLgrAxis objects:

```
; X-axis title.
oTextXAxis = OBJ_NEW('IDLgrText', 'Time (seconds)')
; X-axis (date/time axis).
oPlotXAxis = OBJ_NEW('IDLgrAxis', 0, /EXACT, RANGE = xr, $
   XCOORD_CONV = xs, YCOORD_CONV = ys, TITLE = oTextXAxis, $
   LOCATION = [xr[0], yr[0]], TICKDIR = 0, $
   TICKLEN = (0.02*(yr[1] - yr[0])), $
   TICKFORMAT = ['LABEL_DATE'], TICKINTERVAL = 5, $
   TICKUNITS = ['Time'])
; Y-axis title.
oTextYAxis = OBJ_NEW('IDLgrText', 'Displacement (inches)')
; Y-axis.
oPlotYAxis = OBJ_NEW('IDLgrAxis', 1, /EXACT, RANGE = yr, $
   XCOORD_CONV = xs, YCOORD_CONV = ys, TITLE = oTextYAxis, $
   LOCATION = [xr[0], yr[0]], TICKDIR = 0, $
   TICKLEN = (0.02*(xr[1] - xr[0])))
; Plot title.
oPlotText = OBJ_NEW('IDLgrText', 'Measured Signal', $
   LOCATIONS = [(xr[0] + xr[1])/2., $
      (yr[1] + (0.02*(yr[0] + yr[1])))], $
```

```
XCOORD_CONV = xs, YCOORD_CONV = ys, $
ALIGNMENT = 0.5)
```

The TICKFORMAT, TICKINTERVAL, and TICKUNITS keywords specify the X-axis as a date/time axis.

These objects are now added to the oPlotModel object and this model is added to the oPlotView object:

```
oPlotModel->Add, oPlot
oPlotModel->Add, oPlotXAxis
oPlotModel->Add, oPlotYAxis
oPlotModel->Add, oPlotText
oPlotView->Add, oPlotModel
```

Now the oPlotView object, which contains all of these objects, can be viewed in the oPlotWindow object:

```
oPlotWindow->Draw, oPlotView
```

The Draw method to the oPlotWindow object produces the following results:

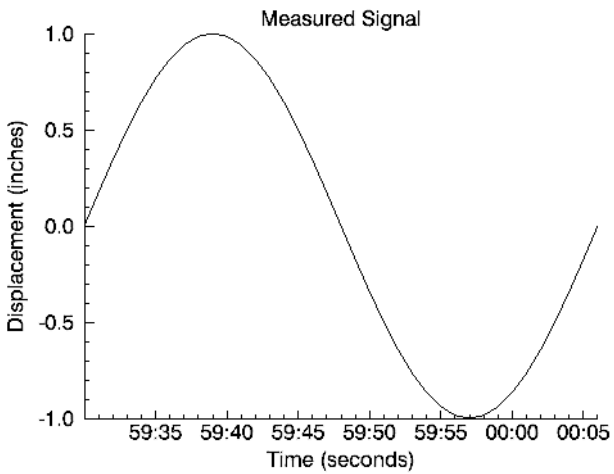


Figure 5-8: Displaying Date/Time data with IDLgrPlot

The above display shows the progression of the date/time variable, but it does not include all of the date/time data we generated with the TIMEGEN routine. This data also includes hour, month, day, and year information. IDL can display this information with additional levels to the date/time axis. You can control the number of levels to draw and the units used at each level with the TICKUNITS keyword. You

can specify the formatting for these levels by changing the DATE_FORMAT keyword setting to the LABEL_DATE routine:

```
date_label = LABEL_DATE(DATE_FORMAT = $
   ['%I:%S', '%H', '%D %M, %Y'])
```

where %H represents hours, %D represents days, %M represents months, and %Y represents years. Notice DATE_FORMAT is specified with a three-element vector. Date/time data can be displayed on an axis with three levels. The format of these levels are specified through this vector.

In this example, the first level (closest to the axis) will contain minute and second values separated by a colon (%I:%S). The second level (just below the first level) will contain the hour values (%H). The third level (the final level farthest from the axis) will contain the day and month values separated by a space and year value separated from the day and month values by a comma (%D %M, %Y). For more information, see LABEL_DATE in the *IDL Reference Guide*.

Besides the above change to the LABEL_DATE routine, we must also change the settings of the IDLgrAxis properties to specify a multiple level axis:

```
oPlotXAxis->SetProperty, $
   TICKFORMAT = ['LABEL_DATE', 'LABEL_DATE', 'LABEL_DATE'], $
   TICKUNITS = ['Time', 'Hour', 'Day']
```

The TICKFORMAT is now set to a string array containing an element for each level of the axis. The TICKUNITS keyword is set to note the unit of each level. These property settings produce the following results:

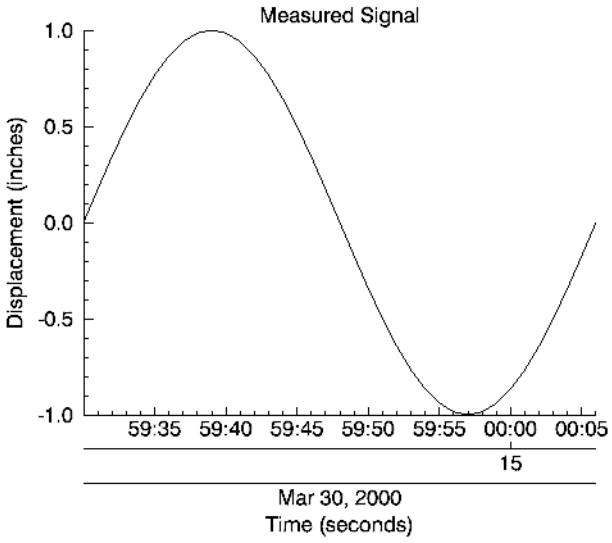


Figure 5-9: Displaying Three Levels of Date/Time data with IDLgrPlot

Notice the three levels of the X-axis. These levels are arranged as specified by the previous call to the LABEL_DATE routine.

To maintain IDL's memory, the object references for oPlotView, oTextXAxis, and oTextYAxis should be destroyed. Therefore, after the display is drawn, the OBJ_DESTROY routine should be called:

```
OBJ_DESTROY, [oPlotView, oTextXAxis, oTextYAxis]
```

The display will remain until closed, but the object references are now freed from IDL's memory.

Displaying Date/Time Data on a Contour Display

Another possible example may be the surface temperature (in degrees Celsius) of each degree of a single circle on a sphere recorded at every second for 37 seconds after the initial recording of 59 minutes and 30 seconds after 2 o'clock pm (14 hundred hours) on the 30th day of March in the year 2000:

```
number_samples = 37
date_time = TIMEGEN(number_samples, UNITS = 'Seconds', $
   START = JULDAY(3, 30, 2000, 14, 59, 30))
angle = 10.*FINDGEN(number_samples)
temperature = BYTSCL(SIN(10.*!DTOR* $
   FINDGEN(number_samples)) # COS(!DTOR*angle))
```

As with the one-dimensional case, the format of the date/time values is specified through the LABEL_DATE routine as follows:

```
date_label = LABEL_DATE(DATE_FORMAT = $
   ['%I:%S', '%H', '%D %M, %Y'])
```

where %I represents minutes, %S represents seconds, %H represents hours, %D represents days, %M represents months, and %Y represents years.

The first level (closest to the axis) will contain minute and second values separated by a colon (%I:%S). The second level (just below the first level) will contain the hour values(%H). The third level (the final level farthest from the axis) will contain the day and month values separated by a space and year value separated from the day and month values by a comma (%D %M, %Y).

Since the final contour display will be filled, we should define a color palette:

```
oContourPalette = OBJ_NEW('IDLgrPalette')
oContourPalette->LoadCT, 5
```

As in the one-dimensional example, the display must be initialized:

```
oContourWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = [800, 600])
oContourView = OBJ_NEW('IDLgrView', /DOUBLE)
oContourModel = OBJ_NEW('IDLgrModel')
oContour = OBJ_NEW('IDLgrContour', temperature, $
   GEOMX = angle, GEOMY = date_time, GEOMZ = 0., $
   /PLANAR, /FILL, PALETTE = oContourPalette, $
   /DOUBLE_GEOM, C_VALUE = BYTSCL(INDGEN(8)), $
   C_COLOR = BYTSCL(INDGEN(8)))
; Applying contour lines over the original contour display.
oContourLines = OBJ_NEW('IDLgrContour', temperature, $
   GEOMX = angle, GEOMY = date_time, GEOMZ = 0.001, $
   /PLANAR, /DOUBLE_GEOM, C_VALUE = BYTSCL(INDGEN(8)))
```

The oContourModel object will contain the IDLgrContour and IDLgrAxis objects. The oContourView object contains the oContourModel with the DOUBLE keyword. The DOUBLE and DOUBLE_GEOM keywords are set for the oContourView and oContour objects because date/time data is made up of double-precision floating-point values.

Although the date/time part of the data will actually be contained and displayed through the IDLgrAxis object, the oContour object is created first to provide a display region for the axes:

```
oContour->GetProperty, XRANGE = xr, YRANGE = yr, ZRange = zr
xs = NORM_COORD(xr)
xs[0] = xs[0] - 0.5
ys = NORM_COORD(yr)
ys[0] = ys[0] - 0.5
oContour->SetProperty, XCOORD_CONV = xs, YCOORD_CONV = ys
oContourLines->SetProperty, XCOORD_CONV = xs, YCOORD_CONV = ys
```

The oContourLines object is created to display contour lines over the filled contours. Note these lines have a GEOMZ difference of 0.001 from the filled contours. This difference is provided to display the lines over the filled contours and not in the same view plane. The NORM_COORD routine is used to create a normalized (0 to 1) display coordinate system. This coordinate system will also apply to the IDLgrAxis objects:

```
; X-axis title.
oTextXAxis = OBJ_NEW('IDLgrText', 'Angle (degrees)')
; X-axis.
oContourXAxis = OBJ_NEW('IDLgrAxis', 0, /EXACT, RANGE = xr, $
   XCOORD_CONV = xs, YCOORD_CONV = ys, TITLE = oTextXAxis, $
   LOCATION = [xr[0], yr[0], zr[0] + 0.001], TICKDIR = 0, $
   TICKLEN = (0.02*(yr[1] - yr[0])))
; Y-axis title.
oTextYAxis = OBJ_NEW('IDLgrText', 'Time (seconds)')
; Y-axis (date/time axis).
oContourYAxis = OBJ_NEW('IDLgrAxis', 1, /EXACT, RANGE = yr, $
   XCOORD_CONV = xs, YCOORD_CONV = ys, TITLE = oTextYAxis, $
   LOCATION = [xr[0], yr[0], zr[0] + 0.001], TICKDIR = 0, $
   TICKLEN = (0.02*(xr[1] - xr[0])), $
   TICKFORMAT = ['LABEL_DATE', 'LABEL_DATE', 'LABEL_DATE'], $
   TICKUNITS = ['Time', 'Hour', 'Day'], $
   TICKLAYOUT = 2)
oContourText = OBJ_NEW('IDLgrText', $
   'Measured Temperature (degrees Celsius)', $
   LOCATIONS = [(xr[0] + xr[1])/2., $
      (yr[1] + (0.02*(yr[0] + yr[1])))], $
   XCOORD_CONV = xs, YCOORD_CONV = ys, $
   ALIGNMENT = 0.5)
```

The TICKFORMAT, TICKINTERVAL, and TICKUNITS keywords specify the Y-axis as a date/time axis, which contains three levels related to the formats presented in the call to the LABEL_DATE routine. This example also contains the TICKLAYOUT keyword. By default, this keyword is set to 0, which provides the date/time layout shown in the plot example. In this example, TICKLAYOUT is set to 2, which rotates and boxes the tick labels.

These objects are now added to the oContourModel object and this model is added to the oContourView object:

```
oContourModel->Add, oContour
oContourModel->Add, oContourLines
oContourModel->Add, oContourXAxis
oContourModel->Add, oContourYAxis
oContourModel->Add, oContourText
oContourView->Add, oContourModel
```

Now the oContourView object, which contains all of these objects, can be viewed in the oContourWindow object:

```
oContourWindow->Draw, oContourView
```

The Draw method to oContourWindow produces the following results:

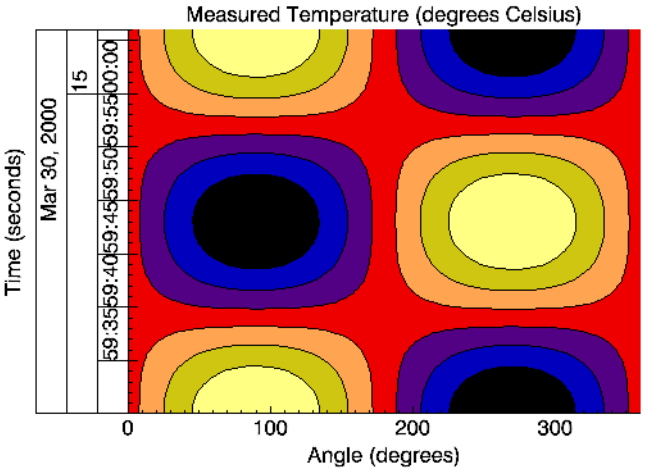


Figure 5-10: Displaying Date/Time data with IDLgrContour

Notice the three levels of the Y-axis. These levels are arranged as specified by the previous call to the LABEL_DATE routine.

To maintain IDL's memory, the object references for oContourView, oContourPalette, oTextXAxis, and oTextYAxis should be destroyed. Therefore, after the display is drawn, the OBJ_DESTROY routine should be called:

```
OBJ_DESTROY, [oContourView, oContourPalette, $
   oTextXAxis, oTextYAxis]
```

The display will remain until closed, but the object references are now freed from IDL's memory.

Axis Titles and Tickmark Text

You can supply an axis title for an axis by setting the TITLE property equal to the object reference of an IDLgrText object. Text objects connected to axis objects via the TITLE property are automatically centered under or next to the axis they belong with.

Note

Titles and tickmark text inherit the color specified for the IDLgrAxis object itself, even if the COLOR property is specified for the IDLgrText object specified, unless the USE_TEXT_COLOR property for the axis is nonzero.

By default, major tick marks are labelled with the data values. You can supply a set of tickmark text values by setting the TICKTEXT property equal to either a single instance of an IDLgrText object containing a vector of text strings or to a vector of IDLgrText objects, each of which contains a single text string.

Note

Make sure that you have the same number of tick label strings as there are major tick marks for the axis.

Reverse Axis Plotting

IDL also allows you to plot data in Object Graphics by reversing the order of axis tick values. This is known as reverse axis plotting.

When using Object Graphics, each core object is a building block. Any number of building blocks may be combined together in a hierarchical tree to create an overall scene. An individual object is not aware of the other objects in the hierarchy; therefore, the designer of the hierarchy must control all interactions between the objects. For example, to properly display a reverse axis plot in Object Graphics, the designer must appropriately set the properties on the X axis, the Y axis, and the plot line, each of which contribute to the overall displayed results.

Example Code

You can run this example by entering `EX_REVERSE_PLOT` at the IDL command line. You can view the source for this example, `ex_reverse_plot.pro`, in the `examples/doc/objects` directory. Run the example procedure by entering `ex_reverse_plot` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT ex_reverse_plot.pro`.

The following figure demonstrates how you can reverse the order of axis tick values using Object Graphics.

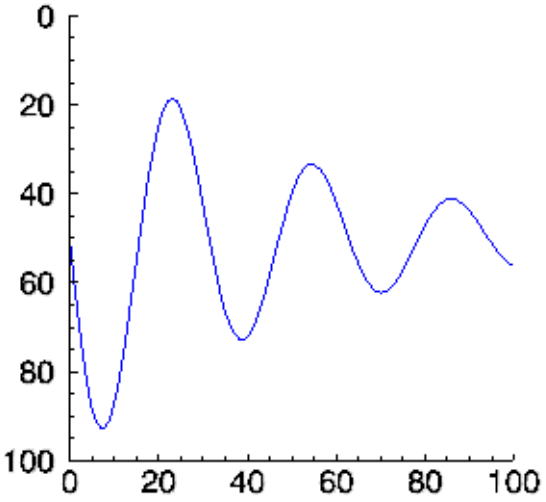


Figure 5-11: Reverse Axis Plotting Example

Symbol Objects

Objects of the IDLgrSymbol class are used to display individual data points, either in an IDLgrPlot object or an IDLgrPolyline object. You can create symbol objects that display one of seven pre-defined symbols, any visualization object, or any model object.

Creating Symbol Objects

Specify the type of symbol to use when you call the IDLgrSymbol::Init method.

To Use a Pre-defined Symbol

Specify one of the following values for the symbol type:

1	Plus sign (the default)
2	Asterisk
3	Period
4	Diamond
5	Triangle
6	Square
7	X

For example, to create a symbol object using a red triangle for the symbol, use the following statement:

```
mySymbol = OBJ_NEW('IDLgrSymbol', 5, COLOR=[255,0,0])
```

To Use a Graphic Object as a Symbol

You can use an visualization object or a model object as a symbol. For best results, create an object that fills the domain between –1 and 1 in all directions. For example, the following statements create a polygon object in the shape of a pentagon and define a symbol object to use the polygon:

```
pentagon=OBJ_NEW('IDLgrPolygon', [-0.8,0.0,0.8,0.4,-0.4], $
    [0.2,0.8,0.2,-0.8,-0.8], COLOR=[0,0,255])
mySymbol = OBJ_NEW('IDLgrSymbol', pentagon)
```

Note that we create the pentagon to fit in the plane between –1 and 1 in both the *X* and *Y* directions. We could also have created the pentagon to fit in a unit square and then scaled it to fit the domain between –1 and 1.

For example:

```
pentagon=OBJ_NEW('IDLgrPolygon', [0.1,0.5,0.9,0.7,0.3], $
   [0.6,0.9,0.6,0.1,0.1], COLOR=[0,0,255])
symModel = OBJ_NEW('IDLgrModel')
symModel->Add, pentagon
symModel->Scale, 2, 2, 1
symModel->Translate, -1, -1, 0
mySymbol = OBJ_NEW('IDLgrSymbol', symModel)
```

Note

We create the symbol object to use the model object rather than the polygon object. Using a model object as a symbol allows you to apply transformations to the symbol even after it has been created.

Setting Size

By default, symbols extend one unit to each side of the data point they represent. Set the SIZE property of the symbol object to a two-element vector that describes the scaling factor in *X* and *Y* to apply to the symbol to change the size of the symbols that are rendered. For example, to scale a symbol so that it extends one tenth of a unit to each side of the data point, use the statement:

```
mySymbol->SetProperty, SIZE=[0.1, 0.1]
```

Setting Color

If you are using a pre-defined symbol, you can set its color using the COLOR property of the symbol object. If you are using a graphic object as a symbol, the symbol's color is determined by the color of the graphic object and the setting of the COLOR property of the symbol object itself is ignored. For example, the following statements create a symbol object that uses a red triangle:

```
mySymbol = OBJ_NEW('IDLgrSymbol', 5, COLOR=[255,0,0])
```

See "IDLgrSymbol" (*IDL Reference Guide*) for details on creating symbol objects.

Using Symbol Objects

To use a symbol, set the SYMBOL property of an IDLgrPlot or IDLgrPolyline object equal to the symbol object reference:

```
myPlot->SetProperty, SYMBOL=mySymbol
```

Suppose you wish to create a symbol object using the pentagon we created above. Suppose also that you wish to be able to use the pentagon code in more than one instance, and would like to be able to make changes to the pentagon object's color, size, and orientation. You might create a procedure to define a pentagon object contained in a model object, and return the object references.

Example Code

See file `penta.pro`, located in the `examples/doc/objects` subdirectory of the IDL distribution to view the source code for this example. Run the example procedure by entering `penta` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT penta.pro`.

Once you have compiled the penta procedure, call it with the SYMBOL and MODEL keywords set equal to named variables that will contain the object references of the model and polygon objects:

```
PENTA, SYMBOL=sym, MODEL=symmodel
```

Next, create a symbol object using the pentagon:

```
mySymbol = OBJ_NEW('IDLgrSymbol', symmodel)
```

Now, create a plot object using the pentagon as the plot symbol:

```
myPlot = OBJ_NEW('IDLgrPlot', FINDGEN(10), SYMBOL=mySymbol)
```

Next, display the plot:

```
myView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0,0,10,10])
myModel = OBJ_NEW('IDLgrModel')
myView->Add, myModel
myModel->Add, myPlot
myWindow = OBJ_NEW('IDLgrWindow')
myWindow->Draw, myView
```

Note that the plotting symbols are larger than you might wish. Try making them smaller:

```
mySymbol->SetProperty, SIZE=[0.2,0.2]
myWindow->Draw, myView
```

Or, create the following procedure to spin the pentagons around the *z*-axis (enter `.RUN` at the command prompt, followed by these statements):

```
PRO SPIN, model, view, window, steps
FOR i = 0, steps do begin
    model->Rotate, [0,0,1], 10
    window->Draw, view
END
END
```

After compiling the SPIN procedure, call it from the command line and watch the pentagons spin:

```
SPIN, symmodel, myView, myWindow, 100
```

While it is unlikely that you will wish to create spinning plot symbols, this example demonstrates one of the key advantages of IDL Object Graphics over IDL Direct Graphics—once created, graphics objects can be easily manipulated in a variety of ways without the need to recreate the entire graph or image after each change.

A Plotting Routine

This section develops a plotting routine that uses many of the object graphics features discussed here and in previous chapters.

Example Code

The code for this example is contained in the file `obj_plot.pro`, located in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `obj_plot` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT obj_plot.pro`.

The OBJ_PLOT routine will create a window object, and display within it a view of a single model object, which will contain a plot object, *x*- and *y*-axis objects, and an *x*-axis title object. It will use the Times Roman font for the axis title.

In creating the procedure, we allow the user to specify the data to be plotted, and we define keyword variables which can return the object references for the view, model, window, axis, and plot objects. This allows the user to manipulate the object tree after it has been created. We also specify the _EXTRA keyword, which allows the user to include other keyword parameters in the call. OBJ_PLOT itself passes any extra keyword parameters only to the plot object, but a more complex program could pass keyword parameters to any of the objects created. The following lines begin the procedure.

Note

See "A Function for Coordinate Conversion" on page 81 for a discussion of the NORM_COORD function used in this example. Also, SET_VIEW is discussed in "Finding an Appropriate View Volume" on page 78. (The files `set_view.pro` and `norm_coord.pro` are included in the `examples/doc/utilities` subdirectory of the IDL distribution. NORM_COORD is also defined in the `obj_plot.pro` file.)

Now, the OBJ_PLOT routine can be called with only the data parameter, if you choose. For example, the statement

```
OBJ_PLOT, FINDGEN(10)
```

creates and displays the object hierarchy with a simple plot line. However, if you do not retrieve the window, view, and other object references via the keywords, there is no way you can interactively modify the plot.

A better way to call OBJ_PLOT would be:

```
OBJ_PLOT, FINDGEN(10), WINDOW=win, VIEW=view, PLOT=plot,
CONTAINER=cont
```

This statement creates the same object hierarchy, but returns the object references for the window, view, and plot objects in named variables. Having access the object references allows you to do things like change the color of the plot:

```
plot->SetProperty, COLOR=[255,255,255]
window->Draw, view
```

enlarge the viewplane rectangle by 10 percent:

```
view->GetProperty, VIEWPLANE_RECT=vr
vr2 = [vr[0]-(vr[0]*0.1), vr[1]-(vr[1]*0.1), $
       vr[2]+(vr[2]*0.1), vr[2]+(vr[2]*0.1)]
view->SetProperty, VIEWPLANE_RECT = vr2
window->Draw, view
```

or just clean it up:

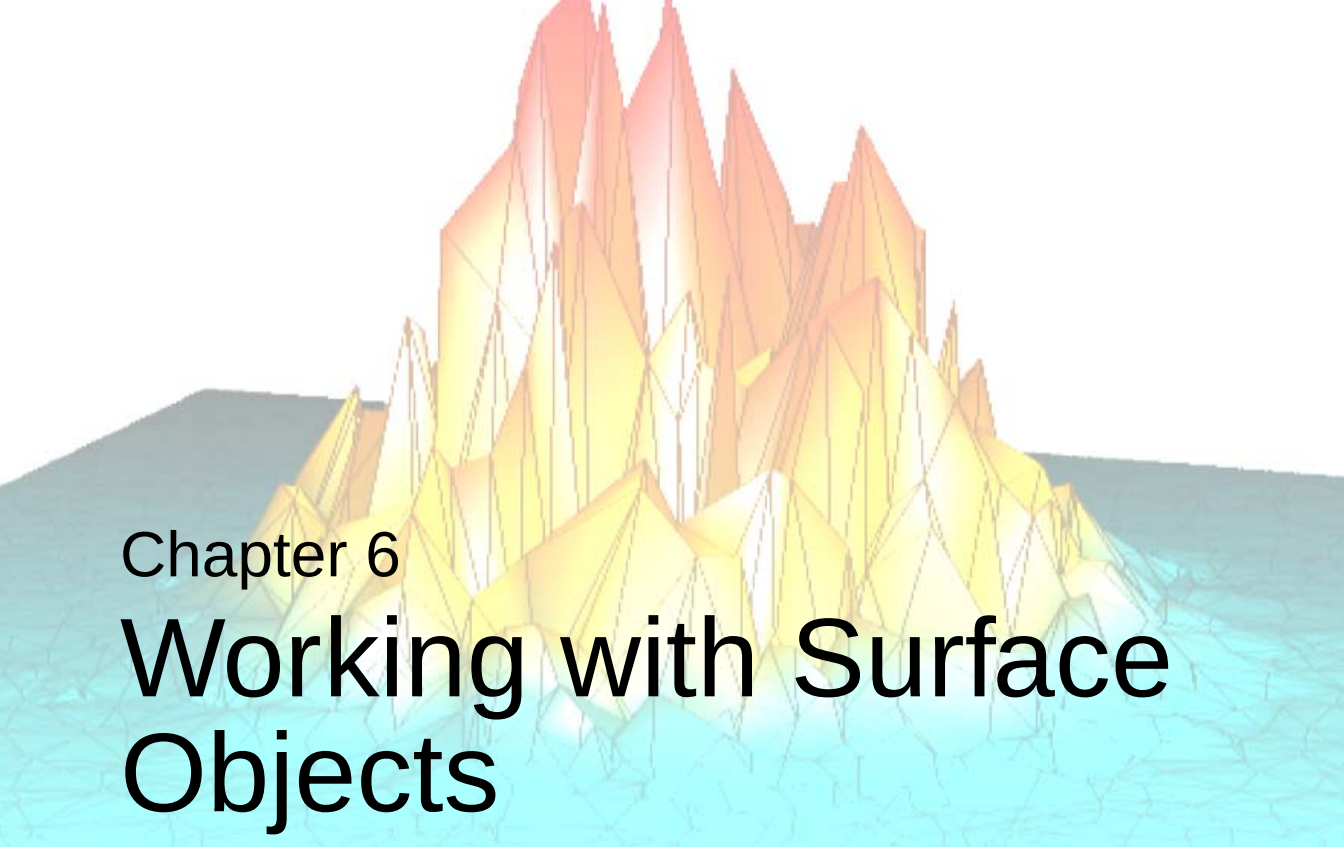
```
OBJ_DESTROY, cont
```

Note that when using the OBJ_DESTROY procedure, any object added to the specified object (using the Add method) are also destroyed, recursively. We use a container object to collect all of the objects, including attribute objects and text object that are not explicitly added to the object tree, which allows you to destroy the entire collection with a single call to OBJ_DESTROY.

Improvements to the OBJ_PLOT Routine

A number of improvements to the OBJ_PLOT routine are left as exercises for the programmer:

- Provide error checking on the input arguments.
- Provide a way to set properties of the axis and text objects when calling `obj_plot`.
- Provide a graphical user interface to using IDL widgets.
- Do the object cleanup (destroying the objects created by `obj_plot`) when the user is finished with the routine. (This is easily accomplished if the routine has a widget interface.)
- Provide a way to retrieve data values once the data has been plotted, using the mouse to select data points.



Chapter 6

Working with Surface Objects

This chapter describes the use of surface and light objects. The following topics are covered in this chapter:

Surface Objects

Surface objects create a representation of functions of two variables. Surfaces are presented as three-dimensional objects in three-dimensional space, and thus are good candidates for interactive rotation, and scaling. Examples in this chapter discuss interactive manipulation of surface objects.

Note

Also see “Mapping an Image onto Elevation Data” (Chapter 3, *Image Processing in IDL*) for additional examples using the surface object.

Creating Surface Objects

To create a surface object, provide a two-dimensional array of surface values (Z values) to the IDLgrSurface::Init method. Optionally, you can supply two vectors or arrays X and Y that specify the locations in the XY plane of the Z values provided. If X and Y are not provided, the surface is generated as a function of the array indices of each element of the Z array.

For example, the following statements create a surface object from the two-dimensional array created by the IDL command DIST, as a function of the Z data array indices:

```
zdata = DIST(40)
mysurf = OBJ_NEW('IDLgrSurface', zdata)
```

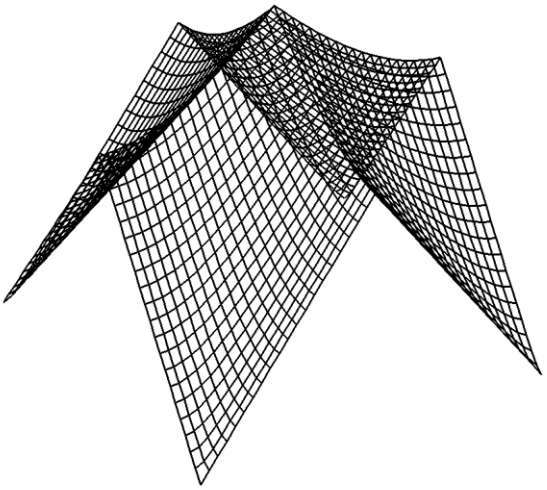


Figure 6-1: Surface Object

Similarly, if xdata and ydata are either 40-element vectors or 40x40 element arrays specifying the X and Y values which, when evaluated by some function, result in the zdata array, you would create the surface object with the following statement:

```
mysurf = OBJ_NEW('IDLgrSurface', zdata, xdata, ydata)
```

See "IDLgrSurface" (*IDL Reference Guide*) for details on creating surface objects.

Using Surface Objects

Surface objects have numerous properties controlling how they are rendered. You can set these properties when creating the surface object, or use the SetProperty method to the surface object to change these properties after creation.

Style

Set the STYLE property to an integer value that controls how the surface is rendered. Set the STYLE property equal to one of the following integer values:

0 = Display a single pixel for each data point.

1 = Display the surface as a wire mesh. (This is the default.)

2 = Display the surface as a solid.

3 = Display the surface using only lines drawn parallel to the *x*-axis.

4 = Display the surface using only lines drawn parallel to the *y*-axis.

5 = Display a wire mesh *lego*-type surface (similar to a histogram plot).

6 = Display a solid *lego*-type surface (similar to a histogram plot).

For example, the following statement changes the surface object to display the surface as a wire mesh, with the lines drawn in blue:

```
mysurf->SetProperty, STYLE=1, COLOR=[0,0,255]
```

The following statement draws the surface as a solid *lego*-type surface in green:

```
mysurf->SetProperty, STYLE=6, COLOR=[0,255,0]
```

Vertex Colors

You can supply a vector of vertex colors via the VERT_COLORS property. The colors in the vector will be applied to each vertex in turn. If there are more vertices than colors supplied for the VERT_COLORS property, IDL will cycle through the colors. For example, the following statements color each vertex and connecting line one of four colors:

```
vcolors =[[0,100,200],[200,150,200],[150,200,250],[250,0,100]]
mysurf->SetProperty, STYLE=1, VERT_COLORS=vcolors
```

Shading

IDL provides two types of shading for surfaces. In Flat shading, the color of the first vertex in the surface is used to define the color for the entire surface. The color has a constant intensity. In Gouraud shading, the colors along each line are interpolated between vertex colors, and then along scanlines from each of the edge intensities.

Note

By default, only ambient lighting is provided for surfaces. If you do not supply a light source for your object hierarchy, solid surface objects will appear flat with either Flat or Gouraud shading. See “Light Objects” on page 233 for details on creating and using light objects.

Set the SHADING property of the surface object equal to 0 (zero) to use flat shading (this is the default), or equal to 1 (one) to use Gouraud shading. In the above example using vertex colors, adding the following statement:

```
mysurf->SetProperty, STYLE=2, SHADING=1
```

creates a surface in which the color values are interpolated between the vertex colors.

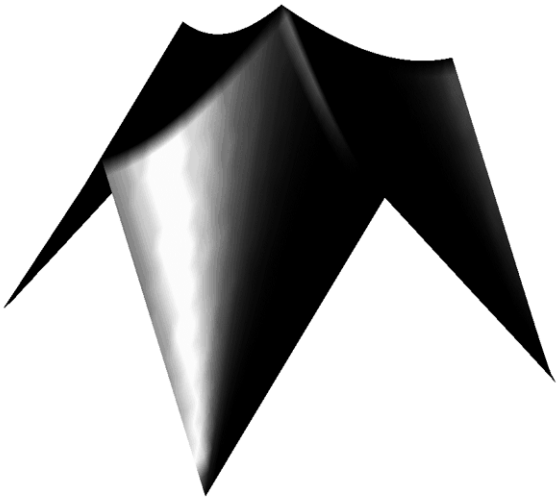


Figure 6-2: Surface Object Shading

Skirts

You can draw a skirt around the bottom edge of your surface object by setting the SHOW_SKIRT property of the surface object to 1. The skirt extends from the edge of the surface to a *Z* value specified by the SKIRT property. For example, the following statements draw the surface in wire mesh mode, with a skirt extending from the bottom of the surface to the value *z* = 0.1:

```
mysurf->SetProperty, STYLE=1, /SHOW_SKIRT, SKIRT=0.1
```

Hidden Line Removal

Set the HIDDEN_LINES property to the surface object equal to one to remove lines that are behind the visible parts of the surface from the rendering. By default, hidden lines are drawn. The following statement alters the surface to remove the hidden lines:

```
mysurf->SetProperty, /HIDDEN_LINES
```

Warning

Hidden line removal can be time-consuming.

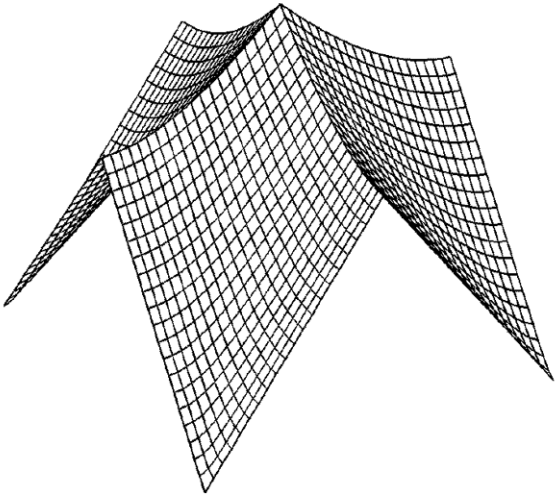


Figure 6-3: Surface Object Hidden Lines

Texture Mapping

You can map an image onto a surface object by specifying an `IDLgrImage` object to the TEXTURE_MAP property. The TEXTURE_COORD property defines how individual data points within the image data are mapped to the surface's vertices. If the TEXTURE_COORD property is not specified, the surface object will map the texture onto the entire data space (the region between 0.0 and 1.0 in normalized coordinates). See Chapter 3, "Mapping an Image onto Geometry" (*Image Processing in IDL*) for examples.

An Interactive Surface Example

With a little programming, we can create an application that allows the user to display a surface object and transform its model tree interactively using the mouse.

Example Code

Example code is located in `surf_track.pro` in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `surf_track` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT surf_track.pro`.

This example uses IDL widgets to create a graphical user interface to an object tree. The SURF_TRACK procedure creates a surface object from user-specified data (or from default data, if none is specified), and places the surface object in an IDL draw widget. The SURF_TRACK interface allows the user to specify several attributes of the object hierarchy via pull-down menus. Finally, the SURF_TRACK procedure uses the example trackball object (see “Interactive 3D Transformations” on page 95 for details) to allow the user to rotate the surface in three dimensions.

Call the SURF_TRACK procedure without an argument to use the default surface (a Bessel function) or with a two-dimensional array as its argument:

```
; Make up some data:
zdata = DIST(40)
SURF_TRACK, zdata
```

We encourage you to inspect the code in `surf_track.pro` for hints on how to create a widget application around a draw widget that uses Object Graphics. Note especially that the SURF_TRACK procedure is well-behaved when it exits, destroying all of the objects it creates so as not to tie up memory with leftover objects for which object references are no longer available.

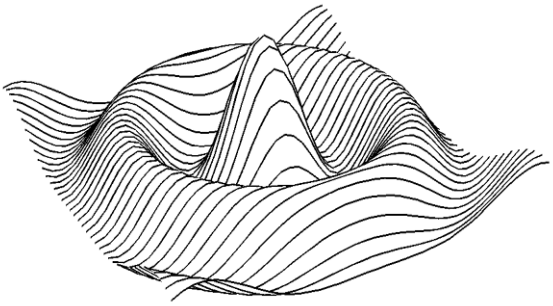


Figure 6-4: STYLE=3 (Ruled xz), HIDDEN_LINES=1 (hidden lines removed)

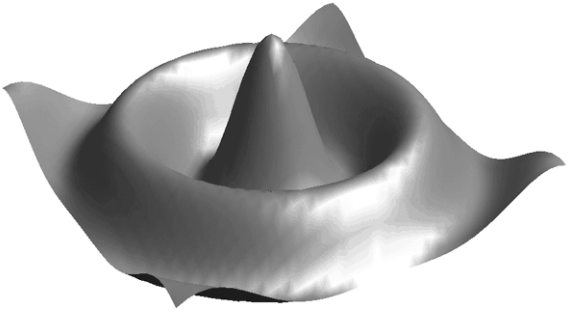


Figure 6-5: SHADING=1 (Gouraud), STYLE=2 (Solid)

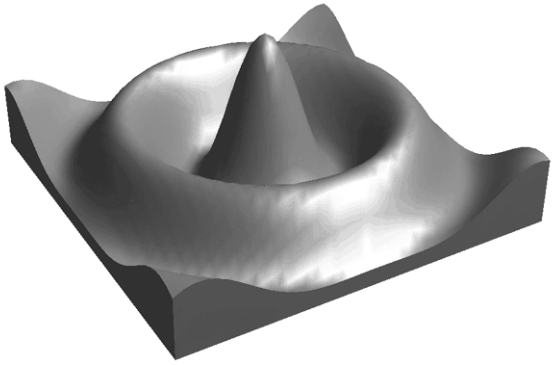
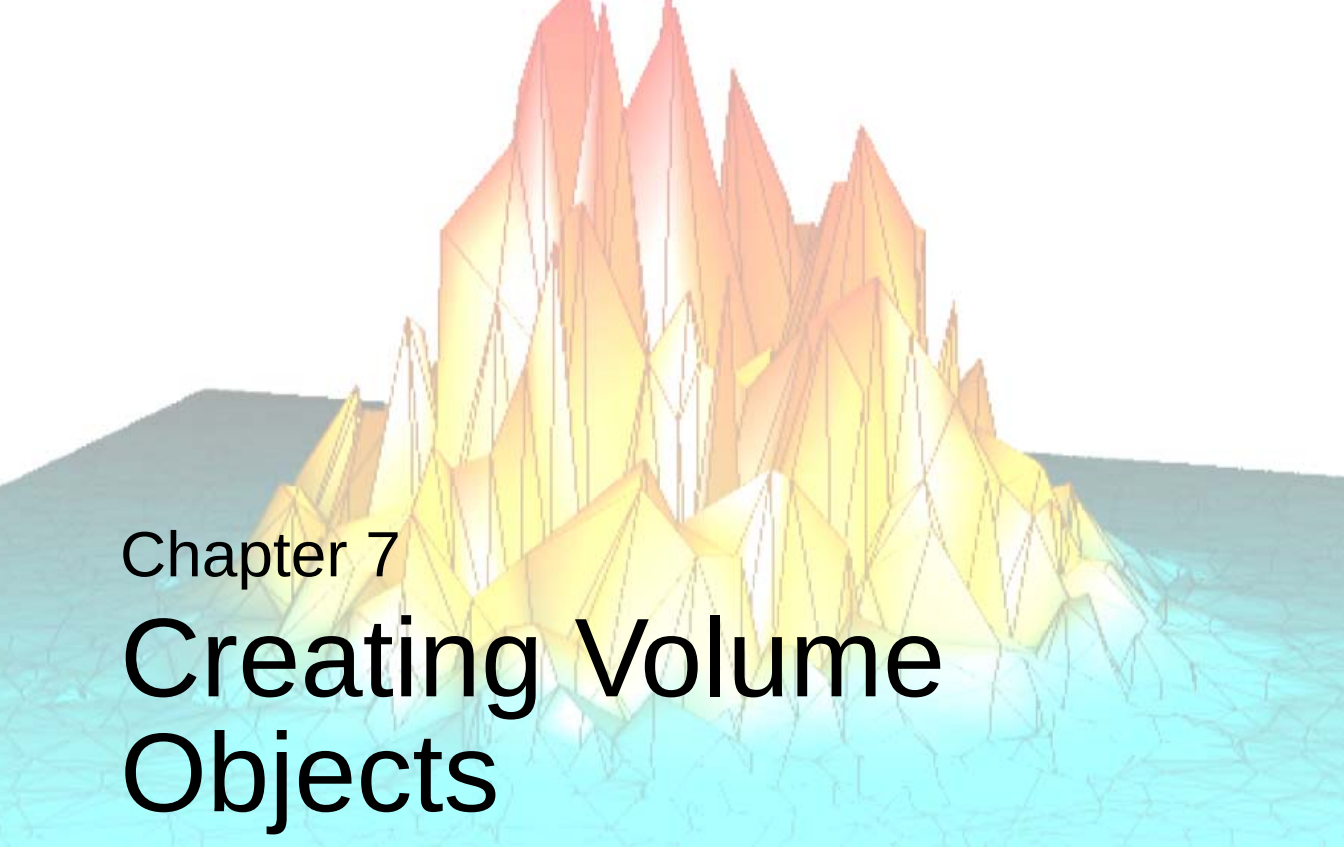


Figure 6-6: SKIRT=-0.402645



Chapter 7

Creating Volume Objects

This chapter describes the process of creating and displaying volume objects. The following topics are covered in this chapter:

Creating a Volume Object

A volume object contains a three dimensional data array of voxel values and a set of rendering attributes. The voxel array is mapped to colors and opacity values through a set of lookup tables in the volume object. Several rendering methods are provided to draw the volume to a destination.

To create a volume object, create a three dimensional array of voxels and pass them to the IDLgrVolume::Init method. Voxel arrays must be of BYTE type. For example, the following will create a simple volume data set and create a volume object which uses it:

```
data = BYTARR(64,64,64, /NOZERO)
FOR i=0,63 DO data[*,i,0:i] = i*2
data[5:15, 5:15, 5:55] = 128
data[45:55, 45:55, 5:15] = 255
myvolume = OBJ_NEW('IDLgrVolume', data)
```

The volume contains a shaded prism along with two brighter cubes (one located within the prism).

See "IDLgrVolume" (*IDL Reference Guide*) for details on creating volume objects.

Example Code

The example code discussed in the following sections is contained in the procedure file `obj_vol.pro`, located in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `.EDIT obj_vol` at the IDL command prompt or view the file in an IDL Editor window by entering `obj_vol.pro`. The procedure file stops after each operation (roughly corresponding to each section below) and requests that you press return before continuing.

Using Volume Objects

A volume object has spatial dimensions equal to the size of the data in the volume. In the example, the volume object occupies the range 0-63 in the *x*-, *y*-, and *z*-axes. To make the volume easier to manipulate, we use the XCOORD_CONV, YCOORD_CONV, and ZCOORD_CONV properties of the volume object to center the volume at 0,0,0 and scale it to fit in a unit cube.

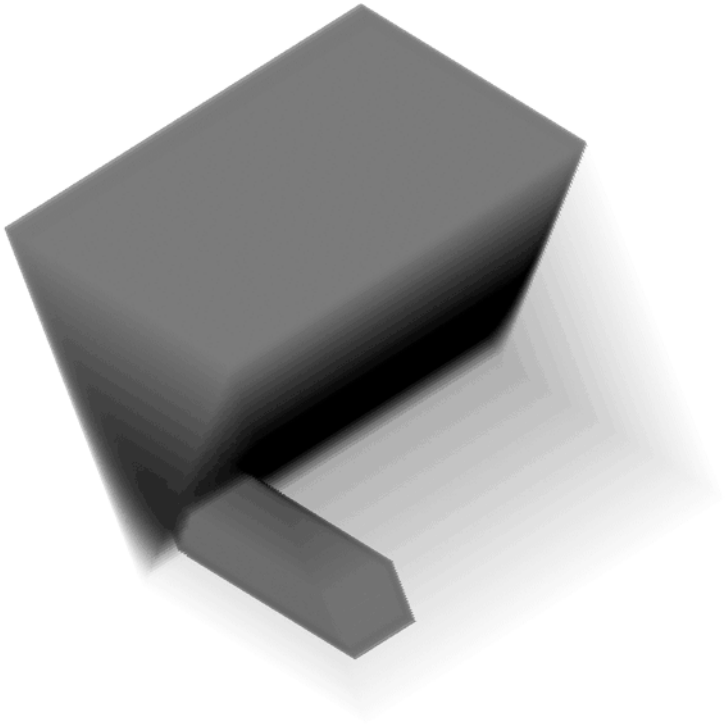


Figure 7-1: Volume Object

Setting Volume Object Attributes

Volume objects have numerous properties controlling how they are rendered. These properties can be set when the object is created or set using the SetProperty method.

Example Code

The example code discussed in the following sections is contained in the procedure file `obj_vol.pro`, located in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `obj_vol` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT obj_vol.pro`. The procedure file stops after each operation (roughly corresponding to each section below) and requests that you press return before continuing.

Volume Opacity

The opacity table controls the transparency of a given voxel value. Manipulation of the opacity table is critical to improving the quality of a rendering. The following figure reflect the sample code, which makes the prism transparent and the cubes opaque, allowing the cube within the prism to be seen. This is done by setting the OPACITY_TABLE0 array to low values for the prism and high values for the cubes.

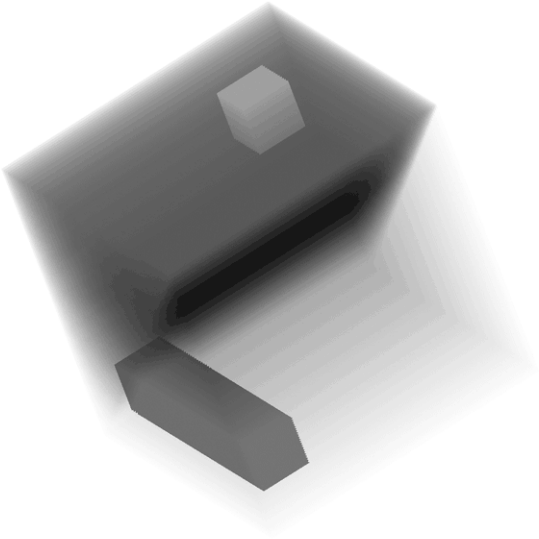


Figure 7-2: Volume Object Opacity

Volume Color

Each voxel value can be assigned an individual color as well. This color mapping can be changed by changing the RGB_TABLE0 property. To further highlight the cubes, we change their colors to blue and red, as shown in the example code, `obj_vol.pro`, located in the `examples/doc/objects` subdirectory of the IDL distribution.

Volume Lighting

Adding lights enhances the edges of volumes. Gradients within the volume are used to approximate a surface normal for each voxel, and the lights in the current view are then applied. The gradient shading is enabled by setting the LIGHTING_MODEL property equal to one. The ambient volume color is selected by setting the AMBIENT property of the volume object to a color value. Setting the TWO_SIDED property allows both sides of a voxel to be lighted. See `obj_vol.pro` in the `examples/doc/objects` subdirectory of the IDL distribution for an example of using a light source.

Note

Only DIRECTIONAL light sources are honored by the volume object. Because normals must be computed for all voxels in a lighted view, enabling light sources increases the rendering time.

See "Light Objects" on page 233 for more details on creating and using light objects.

Compositing

The volume object supports a number of methods for blending the projected voxels together to form an image. By default, Alpha blending is used. (In Alpha blending, each voxel occludes voxels behind it according to the opacity of the voxel in front). Another common compositing technique is the maximum intensity projection (MIP). Set the volume object to use MIP compositing by setting the COMPOSITE_FUNCTION property equal to one as shown in `obj_vol.pro`, located in the `examples/doc/objects` subdirectory of the IDL distribution. See "IDLgrVolume Properties" (*IDL Reference Guide*) for other options.

ZBuffering

When combining a volume with other geometry in the Object Graphics system, volume objects should in general be drawn last to ensure they intersect the other (solid) objects properly. To increase rendering speed, the intersection operation is disabled by default. To enable the intersection calculations, set the ZBUFFER property of the volume object equal to one.

Additionally, volume objects allow for control over the rendering of invisible (opacity equals zero) voxels. By default, the zbuffer will be updated for such voxels (even though no change is made in the image color). This writing to the zbuffer by transparent voxels be disabled by setting the ZERO_OPACITY_SKIP property.

These properties are set near the beginning of the `obj_vol.pro` file, located in the `examples/doc/objects` subdirectory of the IDL distribution.

Note

In volumes with large numbers of voxels with their opacity set to zero, enabling ZERO_OPACITY_SKIP can improve rendering performance.

Interpolation

By default, when rendering a volume object, values between the voxels are estimated using nearest neighbor sampling. When higher quality rendering is desired, trilinear interpolation can be selected instead by setting the INTERPOLATE property equal to one.

```
myvolume->SetProperty, INTERPOLATE=1
```

Note

Trilinear interpolation will cause the rendering to take considerably longer than nearest neighbor interpolation. See "Interpolation Methods" (Chapter 5, *Using IDL*) for more information on interpolation.

Rendering speed

Rendering speed can be improved by reducing the quality of the rendering. Use the RENDER_STEP property to control this speed/quality trade-off. The value of the RENDER_STEP property specifies a step size in the screen dimensions which is used to skip voxels during the rendering process. Larger values yield faster rendering times, but lower final image quality. For example, to render only half as many voxels in the screen Z dimension, use the following statement:

```
myvolume->SetProperty, RENDER_STEP=[1,1,2]
```

A more complex example using a volume object is shown in the volume visualization demo. To start the demos, type demo at the IDL command prompt.

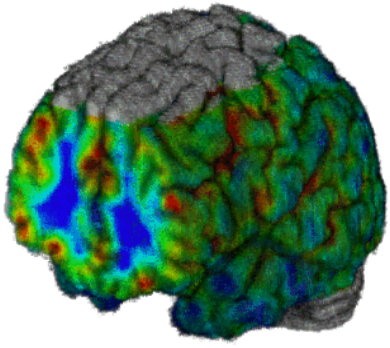
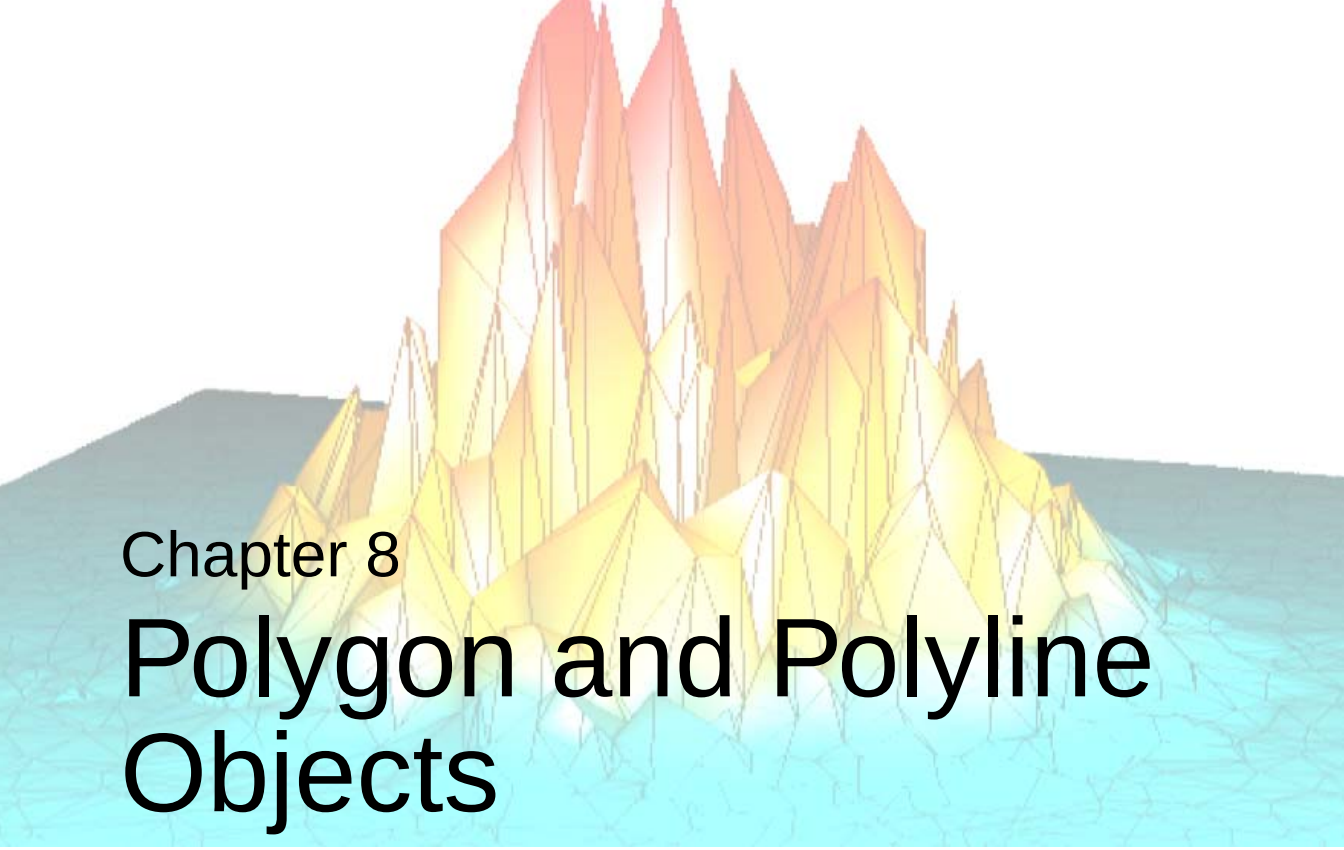


Figure 7-3: Volume Object Rendering



Chapter 8
Polygon and Polyline Objects

This chapter describes the use of polygon, polyline objects. The following topics are covered in this chapter:

About Polygon and Polyline Objects

Polygon and Polyline objects are both defined by set of vertices that share rendering attributes. This chapter introduces how to create and configure polygon and polyline objects.

Creating Polygon and Polyline Objects

You can define the shape of a polygon or polyline object by either setting vertex data directly (by passing a 2-by-*n* or a 3-by-*n* array to the DATA property), or by passing a descriptive array to the IDLgrPolygon POLYGONS property or the IDLgrPolyline POLYLINES property. This section describes the later method, which uses a connectivity array to define the shape of an IDLgrPolygon or IDLgrPolyline object.

Note

The following description of the connectivity array applies to polygons and polylines with the exception that for a polyline mesh, vertex data includes color, but not normals or texture coordinates.

A polygon description is a numeric list of the form: [n, i_0, i_1, ..., i_{n-1}], where n is the number of vertices that define the polygon, and i_0..i_{n-1} are indices into a polygon vertex list. For example, the list [5, 0, 1, 2, 3, 4] describes a polygon with 5 vertices comprised of the first 5 vertices in the vertex list.

The polygon description list, also known as a connectivity array, allows an individual object to contain more than one polygon. The polygons can be independent and distinct, sharing no vertices amongst the polygons. Alternatively, the connectivity array can describe a mesh, where vertices are shared by a number of polygons, usually triangles or quads, in the mesh. In the case of a mesh, the vertex information, including normals, colors, and texture coordinates, is also shared by the polygons composing the mesh. See “Polygon Mesh Optimization” on page 209 for more information.

A polygon description list may contain “skipped” polygon descriptions by replacing a description with zeroes. This may be more convenient than building a new array. For example, if we have a polygon description list containing three triangles:

```
[3, 14, 90, 21, 3, 4, 5, 34, 3, 6, 1, 2]
```

we can skip drawing the middle triangle by setting the array to:

```
[3, 14, 90, 21, 0, 0, 0, 0, 3, 6, 1, 2]
```

The same effect can be achieved by:

```
twoList = [threeList[0:3], threeList[8:11]]
```

A polygon description list can also be terminated early by putting a -1 in the array in the position after the last polygon to be drawn.

```
[3, 14, 90, 21, 3, 4, 5, 34, -1, 6, 1, 2]
```

The -1 at index 8 effectively makes this a list of two polygon descriptions. Entries after the -1 are ignored.

See "Polygon Objects" on page 204 and "Polyline Objects" on page 214 for more information about configuring these object.

Polygon Objects

Polygon objects represent one or more filled polygons that share a given set of vertices and rendering attributes. All polygons must be simple (the edges of the polygon should not intersect) and convex (the shape of the polygon should not have any indentations). Concave polygons can be converted into convex polygons using the helper object IDLgrTessellator. See “Tessellator Objects” on page 206 for more on tessellator objects.

Creating Polygon Objects

To create a polygon object, provide a two- or three-dimensional array (or two or three vectors) containing the locations of the polygon’s vertices to the IDLgrPolygon::Init method. For example, the following statement creates a square with sides one unit in length, with the lower left corner at the origin:

```
mypolygon = OBJ_NEW('IDLgrPolygon', [[0,0], [0,1], [1,1], [1,0]])
```

Setting vertex data upon initialization is the same as using the DATA property. You can also use the POLYGONS property to define the object shape as described in “Creating Polygon and Polyline Objects” on page 202.

See “IDLgrPolygon” (*IDL Reference Guide*) for complete reference information.

Configuring Polygon Objects

Polygon objects have numerous properties controlling how they are rendered. You can set these properties when creating the polygon object, or use the SetProperty method to the polygon object to change these properties after creation.

Style

Set the STYLE property to an integer value that controls how the polygon is rendered. Set the STYLE property equal to 0 (zero) to render only the vertices. The following statement changes the polygon to display only the vertex points, in blue:

```
mypolygon->SetProperty, STYLE=0, COLOR=[0,0,255]
```

Set the STYLE property equal to 1 (one) to render the vertices and lines connecting them. The following statement draws the polygon’s outline in green:

```
mypolygon->SetProperty, STYLE=1, COLOR=[0,255,0,]
```

The default setting for the STYLE property is 2, which produces a filled polygon. The following statement draws the filled polygon in red:

```
mypolygon->SetProperty, STYLE=2, COLOR=[255,0,0]
```

Vertex Colors

You can supply a vector of vertex colors via the VERT_COLORS property. The colors in the vector will be applied to each vertex in turn. If there are more vertices than colors supplied for the VERT_COLORS property, IDL will cycle through the colors. For example, the following statements color each vertex and connecting line one of four colors:

```
vcolors =[[0,100,200],[200,150,200],[150,200,250],[250,0,100]]
mypolygon->SetProperty, STYLE=1, VERT_COLORS=vcolors
```

Fill Patterns

As demonstrated in “Pattern Objects” on page 207, you can fill a polygon with a pattern contained in an IDLgrPattern object. Set the FILL_PATTERN property equal to the object reference of the pattern object. If you have created a pattern object called mypattern, the following statement uses that pattern as the polygon’s fill pattern:

```
mypolygon->SetProperty, STYLE=2, FILL_PATTERN=mypattern
```

Shading

IDL provides two types of shading for filled objects. In Flat shading, the color of the first vertex in each polygon is used to define the color for the entire polygon. The polygon color has a constant intensity. In Gouraud shading, the colors along each line are interpolated between vertex colors, and then along scanlines from each of the edge intensities.

Set the SHADING property of the polygon object equal to 0 (zero) to use flat shading (this is the default), or equal to 1 (one) to use Gouraud shading. In the above example using vertex colors, adding the following statement:

```
mypolygon->SetProperty, STYLE=2, SHADING=1
```

creates a polygon fill in which the color values are interpolated between the vertex colors.

Texture Mapping

You can map an image onto a polygon object by specifying an IDLgrImage object to the TEXTURE_MAP property. The TEXTURE_COORD property defines how individual data points within the image data are mapped to the polygon’s vertices. Note that you must specify both TEXTURE_MAP and TEXTURE_COORD to enable texture mapping.

Tessellator Objects

The IDLgrTessellator class is a helper class that converts a simple concave polygon (or a simple polygon with holes) into a number of simple convex polygons (general triangles). A polygon is simple if it includes no duplicate vertices, if the edges intersect only at vertices, and exactly two edges meet at any vertex.

Tessellation is useful because the IDLgrPolygon object accepts only convex polygons. Using the IDLgrTessellator object, you can convert a concave polygon into a group of convex polygons.

Creating Tessellator Objects

The IDLgrTessellator::Init method takes no arguments. Use the following statement to create a tessellator object:

```
myTess = OBJ_NEW('IDLgrTessellator')
```

See "IDLgrTessellator" (*IDL Reference Guide*) for details on creating tessellator objects.

Using Tessellator Objects

The `obj_tess.pro` procedure creates a concave polygon, attempts to draw it, and then tessellates the polygon and re-draws. Finally, the procedure demonstrates adding a hole to a polygon. (You will be prompted to press Return after each step is displayed.) You can also inspect the source code in the `obj_tess.pro` file for hints on using the tessellator object.

Example Code

The procedure file `obj_tess.pro`, located in the `examples/doc/objects` subdirectory of the IDL distribution, provides an example using the IDLgrTessellator object. Run the example procedure by entering `obj_tess` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT obj_tess.pro`.

Pattern Objects

Objects of the IDLgrPattern class are used to fill objects of the IDLgrPolygon class. Pattern objects can create a solid fill (the default), a line fill (with control over the orientation, spacing, and thickness of the lines used), or a pattern fill (using a byte pattern you specify). Pattern objects do not have a color of their own; patterns take their color from the COLOR property of the polygon they fill.

Creating Pattern Objects

Specify a fill-pattern style when you call the IDLgrPattern::Init method. Set the argument to the Init method equal to zero to create a solid fill, equal to one to create a line pattern, or equal to two to use a bitmap byte array as the fill pattern. For example, the following statement creates a pattern object with a solid fill:

```
myPattern = OBJ_NEW('IDLgrPattern', 0)
```

The following statement creates a pattern object with lines ten pixels apart, 5 pixels wide, at an angle of 30 degrees:

```
myPattern = OBJ_NEW('IDLgrPattern', 1, SPACING=10, THICK=5, $
   ORIENTATION=30)
```

To create a pattern fill, specify a 32-by-4 byte array via the PATTERN property of the pattern object. The byte array you specify will be tiled over the area of the polygon to be filled. For example, the following statements create a pattern fill with a random speckle. The first statement creates a 32-by-4 byte array with random values ranging between 0 and 255. The second statement creates the pattern object.

```
pattern = BYTE(RANDOMN(seed, 32, 4)*255)
myPattern = OBJ_NEW('IDLgrPattern', 2, PATTERN=pattern)
```

See “IDLgrPattern” (*IDL Reference Guide*) for details on creating pattern objects.

Using Pattern Objects

To fill a polygon with the pattern specified by a pattern object, set the FILL_PATTERN property equal to the pattern object reference:

```
myPolygon->SetProperty, FILL_PATTERN = myPattern
```

The following statements create a triangle and fills it with the random speckle pattern:

```
pattern = BYTE(RANDOMN(seed, 32, 4)*255)
myPattern = OBJ_NEW('IDLgrPattern', 2, PATTERN=pattern)
myView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0,0,10,10])
```

```
myModel = OBJ_NEW('IDLgrModel')
myPolygon = OBJ_NEW('IDLgrPolygon', [4, 7, 3], [8, 6, 3],$
   color=[255,0,255], fill_pattern=myPattern)
myView->Add, myModel
myModel->Add, myPolygon
myWindow = OBJ_NEW('IDLgrWindow')
myWindow->Draw, myView
```

Polygon Optimization

Polygon object can be used in a wide variety of graphic displays. Consider consulting the following topics for information on improving the performance of polygon creation and rendering:

- "Polygon Mesh Optimization" on page 209 — describes how to optimize polygon meshes associated with a polygon through the POLYGON keyword
- "Back-face Culling" on page 212 — lets you skip rendering the unseen side of closed polygons
- "Normal Computations" on page 213 — uses normals that can be computed by COMPUTE_MESH_NORMALS instead of the expensive generation of default normals each time a polygon is drawn

Polygon Mesh Optimization

IDLgrPolygon objects consist of a set of vertices and, optionally—via the POLYGON keyword—a connectivity array describing how those vertices are to be connected to form one or more polygons. Internally, IDL can identify three special types of polygonal meshes that may be represented very efficiently and therefore displayed substantially faster than individually described polygons. These special mesh types are characterized by repetitive patterns in the connectivity of the vertices. In performance terms, it is to your advantage to utilize this optimization whenever possible by appropriately preparing the connectivity list according to the rules described for the corresponding type of mesh. The special mesh types are as follows:

- "Quad Strips" on page 210
- "Triangle Fans" on page 211
- "Triangle Strips" on page 211

Quad Strips

A quad strip is a connected set of four-sided polygons. To take advantage of accelerated quad strips, the connectivity should be set up so that the first and last vertex for one quad are the same as the second and third of the previous quad. See the figure below.

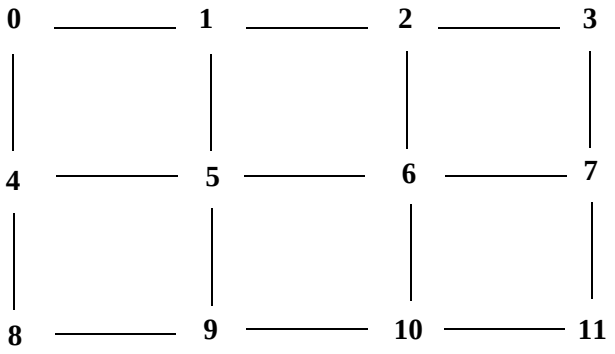


Figure 8-1: Quad Strip Mesh

For example, to use a quad strip optimization for the polygons shown above, the connectivity for the vertices should be as follows:

```
verts = [v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10 ,v11]
oPoly = OBJ_NEW(IDLgrPolygon, verts, $
   POLYGON=[4, 0, 1, 5, 4, $
            4, 1, 2 ,6, 5, $
            4, 2, 3, 7, 6, $
            4, 4, 5, 9, 8, $
            4, 5, 6, 10, 9, $
            4, 6, 7, 11, 10])
```

An alternate connectivity list that still uses quad strip optimization can also be used as in the following example, which orients each quad in the opposite direction of the first example.

```
oPoly = OBJ_NEW('IDLgrPolygon', verts, $
   POLYGON=[4, 4, 5, 1, 0, $
            4, 5, 6, 2, 1, $
            4, 6, 7, 3, 2, $
            4, 8, 9, 5, 4, $
            4, 9, 10, 6, 5,$
            4, 10, 11, 7, 6])
```

Triangle Fans

A triangle fan is a set of connected triangles that all share a common vertex. To take advantage of accelerated triangle fans, the connectivity should be set up so that the first vertex in every triangle is the common vertex, and the second vertex is the same as the last vertex of the previous triangle, as shown below.

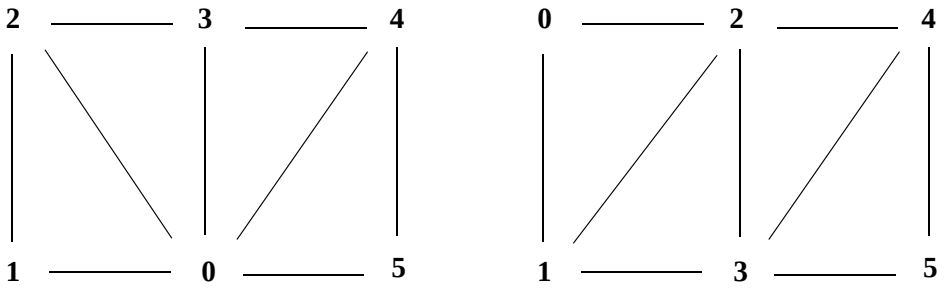


Figure 8-2: Triangle Fan Mesh (left) and Triangle Strip Mesh (right)

For example, to use a triangle fan optimization for the polygons shown in the left side of the figure, the connectivity for the vertices should be as follows:

```
verts = [v0, v1, v2, v3, v4, v5]
oPoly = OBJ_NEW(IDLgrPolygon, verts, $
   POLYGON=[3, 0, 1, 2, $
            3, 0, 2, 3, $
            3, 0, 3, 4, $
            3, 0, 4, 5])
```

Triangle Strips

A triangle strip is a set of connected triangles, each of which share two vertices with the previous triangle. To take advantage of accelerated triangle strips, the connectivity should be set up so that the first two vertices in every triangle must have been in the previous triangle and ordered in the same direction (counter-clockwise or clockwise) and the final vertex must be new, as shown in the right side of the previous figure.

For example, to use the triangle strip optimization for the polygons shown in the right-hand figure, the connectivity for the vertices should be as follows:

```
verts = [v0, v1, v2, v3, v4, v5]
oPoly = OBJ_NEW(IDLgrPolygon, verts, $
   POLYGON=[3, 0, 1, 2, $
            3, 2, 1, 3, $
            3, 2, 3, 4, $
            3, 4, 3, 5])
```

No limits are imposed on the number of meshes or types of meshes within any given polygon object. A single POLYGON keyword value might contain any combination of quad strips, triangle strips, triangle fans, or non-specialized polygons.

As the length of the strips or fans grows, and as the percentage of vertex connections that are optimized by the rules described above increases, the performance upgrade becomes more perceptible. The optimizations are a result of minimizing the time required to perform vertex transforms. If the drawing of the polygons are otherwise limited by fill-rate (as might occur on some systems if texture-mapping is being applied, for instance), these optimizations may not be of significant benefit. In any case, performance will not be hindered in any way by utilizing these specialized meshes, so it is suggested that they be applied whenever possible.

Note

The IDLgrSurface object always takes advantage of the quad mesh optimization automatically without programmer intervention.

Back-face Culling

For polygonal meshes that describe a closed shape (for example, a sphere), it is often wasteful to spend any time rendering the polygons whose normal vector faces away from the eye because it is known that the polygons whose normals face toward the eye will obscure those back-facing polygons. Therefore, for efficiency, it may be beneficial to employ back-face culling, which is simply the process of choosing to skip the rasterization of any polygons whose normal vector faces away from the eye.

On an IDLgrPolygon object, set the REJECT property to a value of 1 to enable back-face culling.

Normal Computations

For IDLgrPolygon objects, normal vectors are computed by default at each vertex by averaging the normals of the polygons that share that vertex. These normals are then used to compute illumination intensities across the surface of the polygon. Computing default normals is a computationally expensive operation. Each time the polygon is drawn, this computation will be repeated if the polygon has changed significantly enough to warrant a new internal cache (for example, if the connectivity, vertices, shading, or style have changed). In some cases, the normals do not actually change as other modifications are made. In these cases, the expense of default normal computation can be bypassed if the user provides the normals explicitly (via the NORMALS keyword). These normals can be computed by using the COMPUTE_MESH_NORMALS routine in the *IDL Reference Guide*. The resulting normals, if passed in via the NORMALS keyword of the IDLgrPolygon object, will be reused every time the polygon is drawn (without further computation) until they are replaced explicitly by the user.

Polyline Objects

Polyline objects lines connect a series of points in two- or three-dimensional space.

Creating Polyline Objects

To create a polyline object, provide a 2-by-*n* or 3-by-*n* array (or two or three vectors) containing the locations of the polyline's constituent points to the IDLgrPolyline::Init method. For example, the following statement creates a line from the origin, to the point *X* = 1, *Y* = 2, then to the point *X* = 4, *Y* = 3:

```
mypolyline = OBJ_NEW('IDLgrPolyline', [[0,0], [1,2], [4,3]])
```

Setting vertex data upon initialization is the same as using the DATA property. You can also use the POLYLINES property to define the object shape as described in "Creating Polygon and Polyline Objects" on page 202.

See "IDLgrPolyline" (*IDL Reference Guide*) for complete reference information.

Using Polyline Objects

Polyline objects have numerous properties controlling how they are rendered. You can set these properties when creating the polyline object, or use the SetProperty method to the polyline object to change these properties after creation.

Symbols

You can specify a symbol to render at each point in the polyline's path by setting the SYMBOL property to the object reference of an IDLgrSymbol object (or to an array of IDLgrSymbol objects). See "Symbol Objects" on page 176 for details.

Shading and Vertex Coloring

Polyline object can be shaded or their vertex points colored in the same manner as polygon objects. See "Shading" and "Vertex Colors" in "Configuring Polygon Objects" on page 204 for details.

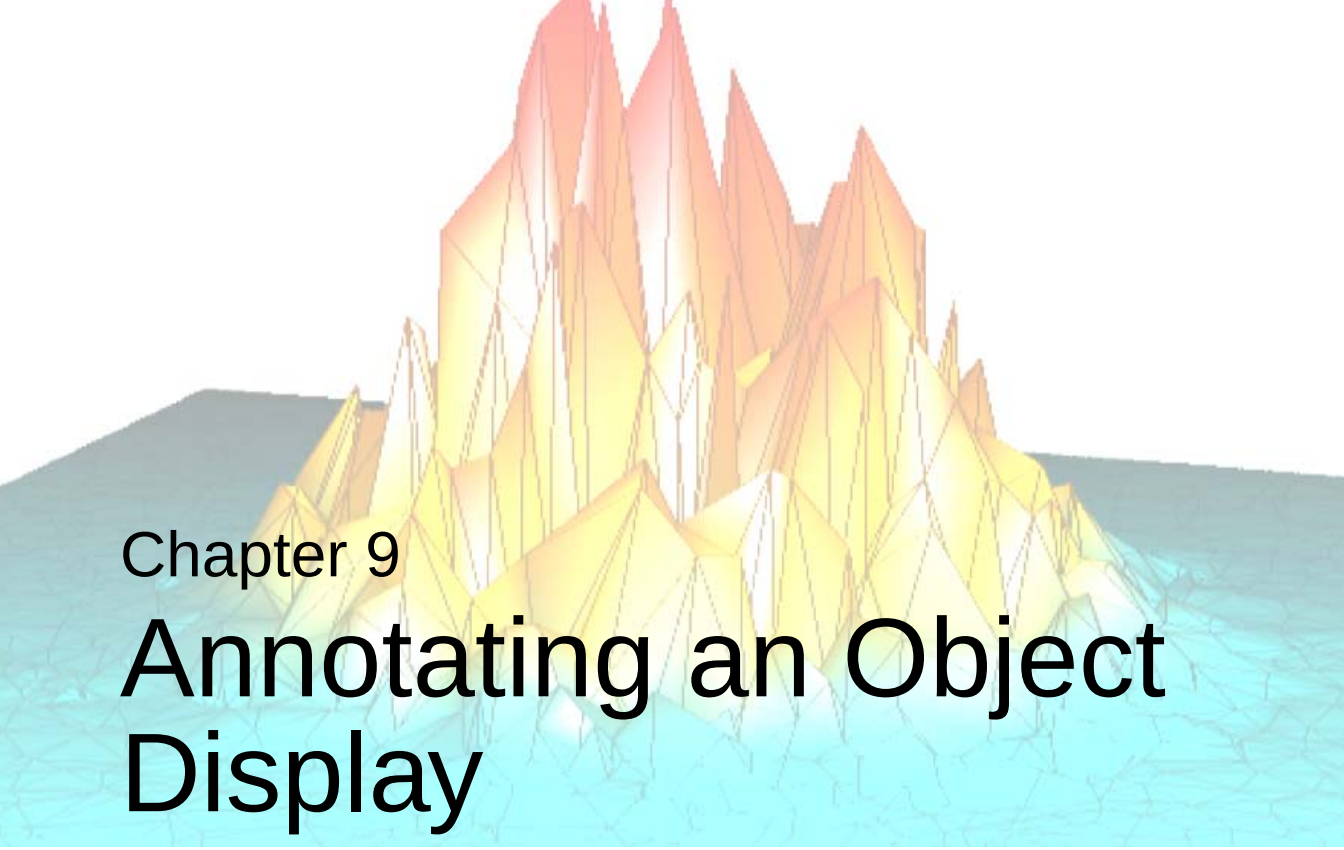
Polygon and Polyline Object Examples

These objects can be used as underlying structures for other objects (such as when texture-mapping an image onto a polygon), or can create an independent 3-dimensional visualization of data as shown in the following examples:

- “Mapping an Image Object onto a Sphere” on page 132
- “Creating a Surface Mesh of an ROI Group” (*Image Processing in IDL*)

Polylines and polygons can also be used in plotting to represent plot data or support the display of plot data as shown in the following examples:

- “DENDROGRAM” (*IDL Reference Guide*) contains an example that uses an IDLgrPolyline
- “Custom Image Object Annotations” on page 236 uses polylines and polygons to construct a custom legend colorbar



Chapter 9

Annotating an Object Display

The following topics are covered in this chapter:

Annotating Object Graphic Displays

Additional objects can be added to the main subjects of an object graphic display (such as a plot, surface, image or volume) to provide explanatory notes or otherwise enhance the information displayed. The objects discussed in this chapter are typically used to further illustrate characteristics of the main subjects of a display. For example, text objects can add descriptive titles, legend objects can distinguish plot data, and light objects can reveal characteristics of surfaces or volumes.

Text Objects

Text objects contain string values that are drawn to the destination object at a location you specify. You have control over the font used (via an IDLgrFont object), the angle of the text baseline, and the vertical direction of the text.

Creating Text Objects

To create a text object, specify a string or an array of strings to the IDLgrText::Init method when calling OBJ_NEW.

```
mytext = OBJ_NEW('IDLgrText', 'A Text String')
```

or

```
mytextarr = OBJ_NEW('IDLgrText', $
   ['First String', 'Second String', 'Third String'])
```

See “IDLgrText” (*IDL Reference Guide*) for details on creating text objects.

Using Text Objects

Creating text annotations in their simplest form—two-dimensional text displayed at a given location—involves only specifying the text, and the location. For example, to display the words Text String in a window in the default font, the following statements suffice:

```
mywindow = OBJ_NEW('IDLgrWindow', DIMENSIONS=[400,400])
myview = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0,0,10,10])
mymodel = OBJ_NEW('IDLgrModel')
mytext = OBJ_NEW('IDLgrText', 'Text String', LOCATION=[4,4], $
   COLOR=[50,100,150])
myview->Add, mymodel
mymodel->Add, mytext
mywindow->Draw, myview
```

The text is drawn at the specified location, with the baseline parallel to the *x*-axis.

Location and Alignment

Specifying a location via the LOCATION property picks a point in space where the text object will be placed. By default, text objects are aligned with their lower left edge located at the point specified by the LOCATION property.

You can change the horizontal position of the text object with respect to the point specified by LOCATION by changing the ALIGNMENT property to a floating-point

value between 0.0 and 1.0. The default value (0.0) aligns and left-justifies text at the location specified. Setting ALIGNMENT to 1.0 right-justifies the text; setting it to 0.5 centers the text above the point specified. The vertical position with respect to location can also be set using the VERTICAL_ALIGNMENT property. The default value (0.0) bottom-justifies the text at the given location. A vertical alignment of 1.0 top-justifies the text.

3D Text and Text “On the Glass”

Text objects, like all graphics atoms, are located and oriented in three-dimensional space. (We often ignore the third dimension when making simple plots and graphs—in these cases we simply use the default *z* value of zero.) With text objects, however, there is an option to project text on the glass.

Projecting text on the glass ensures that it is displayed as if it were in flat, two-dimensional space no matter what its true orientation in three-dimensional space may be. In cases where text objects may be rotated at arbitrary angles, projecting on the glass ensures that the text will be readable.

To project text on the glass, set the ONGLASS property of the text object to a value other than zero.

Figure 9-1: 3D Text and Text “On the Glass”

Baseline

The text baseline can be altered from its default orientation (parallel to the *x*-axis) by setting the text object's BASELINE property to a two- or three-element array. The new baseline will be oriented parallel to a line drawn between the origin and the coordinates specified. For example, the following statement makes the text baseline parallel to a line drawn between the points [0, 0] and [1, 2]:

```
mytext->SetProperty, BASELINE=[1,2]
```

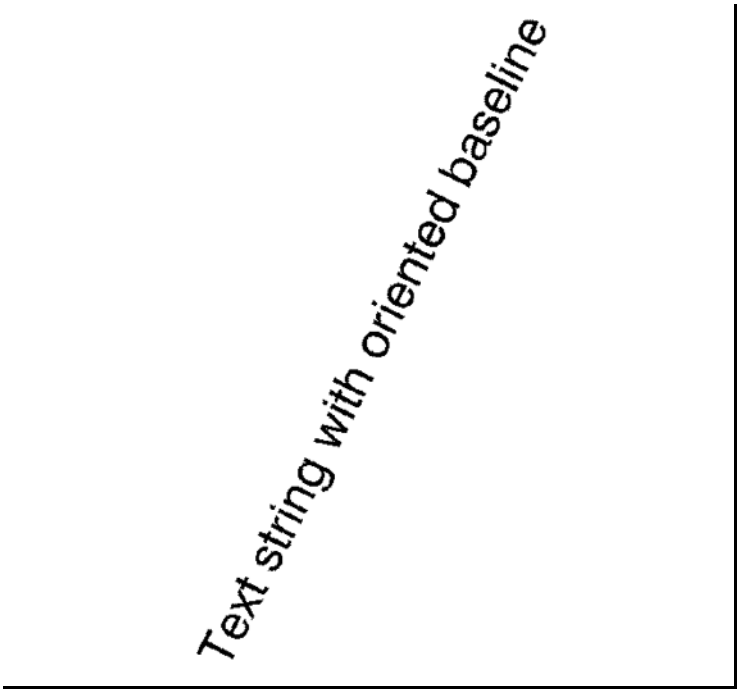


Figure 9-2: Baseline

The following statement makes the baseline parallel to a line drawn between the origin and a point located at [2, 1, 3]:

```
mytext->SetProperty, BASELINE=[2,1,3]
```

Notice that the orientation of the baseline is only an orientation; changing value of the BASELINE property does not change the location of the text object.

Upward Direction

In addition to the baseline orientation, you can control the upward direction of the text object. (The upward direction is the direction defined by a vector pointing from the origin to the point specified.) The upward direction defines the plane on which text is drawn; by specifying a baseline and an upward direction, you define the plane.

Note

The upward direction does not specify a slant angle. That is, even if you specify a direction that is not perpendicular to the baseline for the upward direction, the text will still be perpendicular to the baseline. All that matters is the plane defined by the baseline and upward direction.

For example, in the default situation, the baseline is oriented parallel to the *x*-axis, and the upward direction is parallel to the *y*-axis, pointing in the positive *y* direction.

Warning

If the baseline and upward direction are coincident—that is, if they do not define a plane on which to draw the text—IDL generates an error message.

Fonts

The type style and size of the characters displayed in a text object are controlled by the FONT property. Set the FONT property equal to the object reference of an IDLgrFont object to use that font's properties for the text object. If no font object is specified, IDL uses the default font (12 point Helvetica regular).

Font objects are discussed in "Font Objects" on page 223.

A Text Example

The `rot_text.pro` example creates a simple text string, rotates it around the *y*- and *z*-axes using the BASELINE and UPDIR properties, and displays several different fonts. Also see "Object Graphics Embedded Formatting Examples" on page 225.

Example Code

The procedure `rot_text.pro` is included in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `rot_text` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT rot_text.pro`.

Font Objects

Font objects allow you to specify the type style and size used when rendering objects of the IDLgrText class. You can use either TrueType outline fonts or IDL's built-in Hershey vector fonts. IDL's default font is 12 point Helvetica regular.

Each destination object includes a GetFontnames method, which returns the list of available fonts that can be used in IDLgrFont objects. This method will only return the names of the available TrueType fonts. Hershey vector fonts will not be returned as they are constant—see Appendix H, "Fonts" (*IDL Reference Guide*) for more information. To return all of the TrueType fonts that can be displayed in a window object (`oWindow`), use the following code:

```
fontname=oWindow->GetFontnames("*")
PRINT, fontname
```

See the destination object's GetFontnames method for information on how to return fonts that match specific characteristics.

TrueType Fonts

IDL provides five TrueType outline fonts for use in font objects: Courier, Helvetica, Monospace Symbol, Symbol, and Times. Your system may support additional TrueType fonts —use them in the same way as those supplied by IDL.

A string containing the font name and modifiers defines the characteristics of a font object, as described in "Creating Font Objects" on page 224. The TrueType fonts provided by IDL support the following modifiers:

Font	Modifier
Courier	bold, italic
Helvetica	bold, italic
Monospace Symbol	none
Symbol	none
Times	bold, italic

Table 9-1: TrueType Font Modifiers

Hershey Fonts

IDL supplies a set of vector fonts designed by Dr. A.J. Hershey. See “About Hershey Vector Fonts” (Appendix H, *IDL Reference Guide*) for information on Hershey fonts.

Creating Font Objects

Fonts used by font objects are specified in a string constant constructed from a font name and one or more optional modifiers. When you create a font object, assign the font name string to the NAME property or use it as the IDLgrFont::Init *Fontname* argument. See the following sections for an introduction to creating and configuring font objects. See “IDLgrFont” (*IDL Reference Guide*) for all available options when creating font objects.

Specifying a TrueType Font

The font name is the name by which your computer system knows the font (Times for the Times Roman font, for example). Modifiers specify the weight, angle, and other attributes of the font (bold specifies a weight, italic an angle). The font name string looks like this:

```
'fontname*weight*angle*other_modifiers'
```

where *other_modifiers* can be any other font property supported by a given font, such as a slant. For example, the font name string for Helvetica bold italic is:

```
'helvetica*bold*italic'
```

The font name string for Times Roman Regular is:

```
'times'
```

While the font name must come first in the font name string, the order in which the modifiers are specified is not important. The following statement creates a font object using a bold version of the Times Roman font, with a size of 20 points by replacing the *Fontname* argument with `'times*bold'`:

```
myFont = OBJ_NEW('IDLgrFont', 'times*bold', SIZE=20)
```

See “TrueType Fonts” on page 223 for a list of supported modifiers.

Specifying a Hershey Vector Font

To create a font object using a vector Hershey font, use a string of the format `Hershey*`*`fontnum`* where *`fontnum`* is the Hershey font’s index number. The following statement creates a font object using the Duplex Roman Hershey font, with a size of 14 points:

```
myHersheyFont = OBJ_NEW('IDLgrFont', NAME='hershey*5', SIZE=14)
```

See "Hershey Vector Font Samples" (Appendix H, *IDL Reference Guide*) for descriptions of the Hershey fonts shipped with IDL.

Assigning a Font Object to a Text Object

To use a font object, use the FONT keyword to the IDLgrText::Init method (or change the text object's font via the SetProperty method):

```
myText = OBJ_NEW('IDLgrText', 'Ay, Carumba', FONT = myFont)
```

or

```
myText->SetProperty, STRING='Angstrom symbol: ' + STRING("305B), $
   FONT=myHersheyFont
```

This last example prints the Angstrom symbol by specifying an octal code. See "ISO Latin 1 Encoding" (Appendix H, *IDL Reference Guide*) for details.

If no font object is specified, IDL uses the default font—12 point Helvetica.

Object Graphics Embedded Formatting Examples

Embedded formatting commands are in-line commands that allow you to position text and change fonts within a single line of text. The following examples use both the positioning commands and the font selection commands. All available embedded formatting commands are listed in "Embedded Formatting Commands" (Appendix H, *IDL Reference Guide*).

Tip

Set the ENABLE_FORMATTING property on the IDLgrText object to use formatting commands in Object Graphics.

For example, the following lines of code produce the same output as the Direct Graphics example output shown in "Formatting Command Examples" (Appendix H, *IDL Reference Guide*). This example applies embedded formatting commands that control text positioning.

```
oText = OBJ_NEW('IDLgrText', /ENABLE_FORMATTING)
oText->SetProperty, STRING='!LLower!S!EExponent!R!IIndex' + $
   '!N  Normal!S!EExp!R!IInd!N!S!U Up' + $
   ' !R!D Down!N!S!A Above!R!B Below'
XOBJVIEW, oText
```

You can also change what fonts are used within the text string. For example, you can use the special math symbols available in the Hershey vector font character set (Font 9). When you use the !M formatting command, this applies the font change to the

single character immediately following the !M. Subsequent characters return to the preceding font. The following example produces the same equation as that shown in "A Complex Equation" (Appendix H, *IDL Reference Guide*).

```
; String to produce equation:
SS = '!6F(s) = (2!4p)!e-1/2!n !mi!s!a!e!m' + STRING("44B) +$
      '!r!b!i ' + '-!m' + STRING("44B) + $
      '!nF(x)e !e-i2!4p!3xs!ndx'
myHersheyFont = OBJ_NEW('IDLgrFont', NAME='hershey*5', SIZE=24)
otext = OBJ_NEW('IDLgrText', /ENABLE_FORMATTING)
oText->SetProperty, STRING=ss, FONT=myHersheyFont
XOBJVIEW, oText
```

The font object in this example must use a Hershey font to create the desired results. If no font is specified, the default 12 point Helvetica (not a vector font) is used, and the formatting commands create a different result. See "Changing Fonts within a String" (Appendix H, *IDL Reference Guide*), which defines how formatting commands are applied to Hershey vector and TrueType fonts.

See "Text Objects" on page 219 for details on creating Text objects.

Font Objects and Resource Use

Because font objects are relatively complex, each font object uses a relatively large amount of system resources. As a result, it is better to re-use an existing font object than to create a second identical font object.

ROI Objects

A region of interest (ROI) is an area of an image defined for further analysis or processing. ROIs can be defined programmatically and interactively. The XROI utility lets you interactively define single or multiple regions from an image using the mouse. The utility displays defined ROIs and can output ROI data to specified ROI objects. Any ROI object, whether defined programmatically or interactively, can undergo further processing as an analysis-oriented IDLanROI object, or can be used for display as an IDLgrROI object.

See "Regions of Interest" under the functional category, "Image Processing" (*IDL Quick Reference*) for a list or ROI creation and manipulation routines. Also see "Working with Regions of Interest (ROIs)" (*Image Processing in IDL*) for extensive examples.

Legend Objects

Legend objects provide a simple interface for displaying legends. The legend itself consists of a (filled and/or framed) box around one or more legend items (arranged in a single column) and an optional title string. Each legend item consists of a glyph patch positioned to the left of a text string. The glyph patch is drawn in a square which is a fraction of the legend label font height.

Creating Legend Objects

To create a legend object, you must provide an array of item names, along with arrays of symbols, line styles, or objects, along with arrays of attributes (such as color or thickness) for the items. The following simple example creates a legend object with two items. The first item (Cows) is represented by the predefined symbol number four (a diamond), and the second item (Weasels) is represented by a line-filled box.

```
itemNameArr = ['Cows', 'Weasels']
mytitle = OBJ_NEW('IDLgrText', 'My Legend')
mysymbol = OBJ_NEW('IDLgrSymbol', 4)
mypattern = OBJ_NEW('IDLgrPattern', 1)
myLegend = OBJ_NEW('IDLgrLegend', itemNameArr, TITLE=mytitle, $
   ITEM_TYPE=[0,1], ITEM_OBJECT=[mysymbol, mypattern], $
   /SHOW_OUTLINE)
```

See “IDLgrLegend” (*IDL Reference Guide*) for details on creating legend objects. See the next section for a more detailed explanation of the elements of the legend.

Using Legend Objects

The legend object allows you to define the annotations that correspond to the array of strings used as legend names in a variety of ways. The length of the argument string array is used to determine the number of items to be displayed. Each item is defined by taking one element from the ITEM_NAME, ITEM_TYPE, ITEM_LINESTYLE, ITEM_THICK, ITEM_COLOR, and ITEM_OBJECT vectors, if they are defined. If the number of items (as defined by the argument array or the ITEM_NAME array) exceeds any of the attribute vectors, the attribute defaults will be used for any additional items.

Specify a list of item names either via the argument to IDLgrLegend::Init, or via the ITEM_NAME property. The length of this array determines the size of the legend.

Use the ITEM_TYPE property to define whether an element in the legend is represented by a line (with an optional plotting symbol) or by a filled or unfilled box.

There should be one element of the ITEM_TYPE array per element in the input array or ITEM_NAME array.

Use the ITEM_LINESTYLE and ITEM_THICK properties to define the style and thickness of lines used as legend items. These arrays are ignored for elements that are not lines. Use the ITEM_COLOR property to specify the color of each legend element independently.

Use the ITEM_OBJECT property to specify that a graphic object be used as an annotation.

Dimensions

Until the legend is drawn to the destination object, the [XYZ]RANGE properties will be zero. Because you must know the size of the legend object in order to scale it properly for your window, you must use the ComputeDimensions method on the legend object to get the data dimensions of the legend prior to a draw operation.

The following example builds and displays a three-element legend.

```
; Create a window, view, and model:
mywindow = OBJ_NEW('IDLgrWindow')
myview = OBJ_NEW('IDLgrView')
mymodel = OBJ_NEW('IDLgrModel')
myview->Add, mymodel
; Create the legend with two items:
itemNameArr = ['Original Data', 'Histogram Plot', $
   'Boxcar-filtered (Width=5)']
mytitle = OBJ_NEW('IDLgrText', 'Plot Legend')
mysymbol = OBJ_NEW('IDLgrSymbol', 5, SIZE=[0.3, 0.3])
myLegend = OBJ_NEW('IDLgrLegend', itemNameArr, TITLE=mytitle, $
   BORDER_GAP=0.8, GAP=0.5, $
   ITEM_TYPE=[0,1], ITEM_LINESTYLE=[0,4,2], $
   ITEM_OBJECT=[mysymbol, OBJ_NEW(), OBJ_NEW()], $
   GLYPH_WIDTH=2.0, /SHOW_OUTLINE)
; Add the legend to the model:
mymodel->Add, mylegend
; Center the legend in the window.
; Note that you must use the ComputeDimensions method
; to get the dimensions of the legend.
dims = mylegend->ComputeDimensions(mywindow)
mymodel->Translate, -(dims[0]/2.), -(dims[1]/2.), 0
; Draw the legend:
mywindow->Draw, myview
```

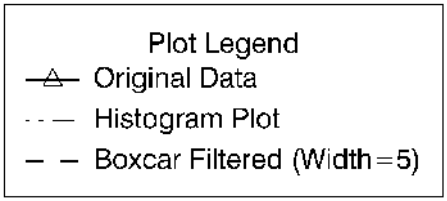


Figure 9-3: Legend Object

Colorbar Objects

The IDLgrColorbar object consists of a color-ramp with an optional framing box and annotation axis. The object can be horizontal or vertical.

Creating Colorbar Objects

To create a colorbar object, you must provide a set of red, green, and blue values to be displayed in the bar. Axis values are determined from the number of elements in the color arrays unless otherwise specified via the TICKVALUES property. The following creates a colorbar one tenth of the window dimension wide by four-tenths of the window dimension high, with a red-green-blue color ramp:

```
mytitle = OBJ_NEW('IDLgrText', 'My Colorbar')
barDims = [0.1, 0.4]
redValues = BINDGEN(256)
greenValues = redValues
blueValues = REVERSE(redValues)
mycolorbar = OBJ_NEW('IDLgrColorbar', redValues, $
   greenValues, blueValues, TITLE=mytitle, $
   DIMENSIONS=barDims, /SHOW_AXIS, /SHOW_OUTLINE)
```

See "IDLgrColorbar" (*IDL Reference Guide*) for details on creating colorbar objects. See the next section for a more detailed explanation of the elements of the legend.

Using Colorbar Objects

The colorbar object allows you to define the size, colors, and various annotations.

Dimensions

Until the legend is drawn to the destination object, the [XYZ]RANGE properties will be zero. Because you must know the size of the legend object in order to scale it properly for your window, you must use the ComputeDimensions method on the legend object to get the data dimensions of the legend prior to a draw operation.

The following example builds and displays the colorbar described above:

```
; Create a window, view, and model:
mywindow = OBJ_NEW('IDLgrWindow')
myview = OBJ_NEW('IDLgrView')
mymodel = OBJ_NEW('IDLgrModel')
myview->Add, mymodel
; Create the colorbar. Make the bar one tenth of
; the window size horizontally and four tenths of
; the window size vertically. Show the axis values (using the
```

```
; default axis annotations) and draw an outline around the bar.
mytitle = OBJ_NEW('IDLgrText', 'My Colorbar')
barDims = [0.1, 0.4]
redValues = BINDGEN(256)
greenValues = redValues
blueValues = REVERSE(redValues)
mycolorbar = OBJ_NEW('IDLgrColorbar', redValues, $
   greenValues, blueValues, TITLE=mytitle, $
   DIMENSIONS=barDims, /SHOW_AXIS, /SHOW_OUTLINE)
mymodel->Add, mycolorbar
; Center the colorbar in the window.
; Note that you must use the ComputeDimensions method to
; get the dimensions of the colorbar.
barPlusTextDims = mycolorbar->ComputeDimensions(mywindow)
mymodel->Translate, -barDims[0]+(barPlusTextDims[0]/2.), $
   -barDims[1]+(barPlusTextDims[1]/2.), 0
; Draw the colorbar:
mywindow->Draw, myview
```

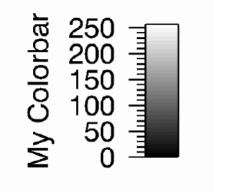


Figure 9-4: Colorbar Object

For more examples of IDLgrColorbar use, see "Displaying Indexed Images with Object Graphics" in the Examples section of "IDLgrPalette::Init" (*IDL Reference Guide*).

Also see "Custom Image Object Annotations" on page 236 for information on configuring a colorbar legend using IDLgrPolygon, IDLgrPolyline and IDLgrText objects.

Light Objects

Objects of the IDLgrLight class represent sources of illumination for graphic objects. Although light objects are not rendered themselves, they are part of the model tree and thus can be transformed along with the graphic objects they illuminate.

If no light sources are specified for a given model, a default ambient light source is supplied. This allows you to display many objects without explicitly creating a light source. The use of only ambient light becomes problematic, however, when solid surfaces and other objects constructed from polygons are displayed. With only ambient lighting, all solid surfaces appear flat—in fact, they appear to be single two-dimensional polygons rather than objects in three-dimensional space.

Note

Graphic objects do not automatically cast shadows onto other objects.

Creating Light Objects

There are no arguments to the IDLgrLight::Init method. Keywords to the Init method allow you to control a number of properties of the light object, including the attenuation, color, cone angle (area of coverage), direction, focus, intensity, location, and type of light.

The following statement creates a default light object. The default light object is a white positional light, located at the origin.

```
mylight = OBJ_NEW('IDLgrLight')
```

There are four types of light objects available. Set the TYPE property of the light object to one of the following integer values:

- 0 = Ambient light. An ambient light is a universal light source, which has no direction or position. An ambient light illuminates every surface in the scene equally, which means that no edges are made visible by contrast. Ambient lights control the overall brightness and color of the entire scene. If no value is specified for the TYPE property, an ambient light is created.
- 1 = Positional light. A positional light supplies divergent light rays, and will make the edges of surfaces visible by contrast if properly positioned. A positional light source can be located anywhere in the scene.
- 2 = Directional light. A directional light supplies parallel light rays. The effect is that of a positional light source located at an infinite distance from scene.

- 3 = Spot light. A spot light illuminates only a specific area defined by the light's position, direction, and the cone angle, or angle which the spotlight covers.

See "IDLgrLight" (*IDL Reference Guide*) for details on creating light objects.

Configuring Light Objects

In addition to the type of light source, you can control several other properties of a light object. The following example creates a solid surface object and displays it first with only ambient lighting, then adds various light objects to the scene.

Note

The SET_VIEW function is discussed in "Finding an Appropriate View Volume" on page 78.

Begin by creating some data, the surface object, and supporting objects:

```
zdata = DIST(40)

mywindow = OBJ_NEW('IDLgrWindow')
myview = OBJ_NEW('IDLgrView')
mymodel = OBJ_NEW('IDLgrMODEL')
mysurf = OBJ_NEW('IDLgrSurface', zdata, STYLE=2)

; Create the object hierarchy:
myview->Add, mymodel
mymodel->Add, mysurf

; Retrieve the X, Y, and Z ranges from the surface object:
mysurf->GetProperty, XRANGE=xr, YRANGE=yr, ZRANGE=zr

; Convert x, y, and z ranges to normalized coordinates.
xnorm = [-xr[0]/(xr[1]-xr[0]), 1/(xr[1]-xr[0])]
ynorm = [-yr[0]/(yr[1]-yr[0]), 1/(yr[1]-yr[0])]
znorm = [-zr[0]/(zr[1]-zr[0]), 1/(zr[1]-zr[0])]

mysurf->SETPROPERTY, XCOORD_CONV=xnorm, $
YCOORD_CONV=ynorm, ZCOORD_CONV=znorm

; Rotate the surface to a convenient orientation:
mymodel->Rotate, [1,0,0], -90
mymodel->Rotate, [0,1,0], 30
mymodel->Rotate, [1,0,0], 30

; Use the SET_VIEW routine to set an appropriate viewplane
; rectangle and zclip region for the view:
```

```
SET_VIEW, myview, mywindow

; Draw the contents of the view:
mywindow->Draw, myview
```

Once the surface object is drawn, we see that there is no definition or apparent three-dimensional shape to the surface. If we add a positional light one unit in the Z direction above the XY origin, however, details appear:

```
mylight = OBJ_NEW('IDLgrLight', TYPE=1, LOCATION=[0,0,1])
mymodel->Add, mylight
mywindow->Draw, myview
```

We can continue to alter the lighting characteristics by changing the properties of the existing light or by adding more light objects. (You can have up to eight lights in a given view object.) We can change the color:

```
mylight->SetProperty, COLOR=[200,0,200]
mywindow->Draw, myview
```

We can change the intensity of the light:

```
mylight->SetProperty, INTENSITY=0.7
mywindow->Draw, myview
```

Note

Also see “Volume Lighting” on page 197 for volume object specific lighting information.

Optimizing Light Object Use

Lighting computations are generally set up to compute the light intensity based on the normal vector for the polygon. If the polygon normal faces away from the eye, the lighting model will likely determine that the light intensity for that polygon is zero. When the polygonal mesh being rendered is a closed surface, this is not a problem because the back-facing polygons will always be obscured. However, when the polygon mesh represents an open shape (for which back-facing polygons may be visible), the dark appearance of these polygons may hinder the user’s perception of the overall shape. In such a case, two-sided lighting can be useful. Two-sided lighting is the process of reversing the normals for all back-facing polygons before computing the light intensities for that polygon.

In IDL’s Object Graphics, two-sided lighting is enabled by default. When the additional lighting calculation is not required, one-sided lighting can be used to improve rendering performance. On an IDLgrModel object, set the LIGHTING property to a value of 1 to enable one-sided lighting.

Custom Image Object Annotations

Many images are annotated to explain certain features or highlight specific details. Color annotations are more noticeable than plain black or white annotations. This section includes the following examples:

- “Annotating Indexed Image Objects”
- “Annotating RGB Image Objects” on page 240

Annotating Indexed Image Objects

When using Object Graphics, the original color table does not need to be modified. The color table (palette) pertains only to the image object not the window, view, model, polygon, or text objects. Color annotations are usually applied to label each color level within the image or to allow color comparisons. This section shows how to label each color level on an indexed image in Object Graphics. As an example, an image of average world temperature is imported from the `worldtmp.png` file. This file does not contain a color table associated with this image, so a pre-defined color table will be applied. This table provides the colors for the polygons and text used to make a colorbar for this image. Each polygon uses the color of each level in the table. The text represents the average temperature (in Celsius) of each level. Complete the following steps for a detailed description of the process.

Example Code

See `applycolorbar_indexed_object.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for code that duplicates this example. Run the example procedure by entering `applycolorbar_indexed_object` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT applycolorbar_indexed_object.pro`.

1. Determine the path to the `worldtmp.png` file:

```
worldtmpFile = FILEPATH('worldtmp.png', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])
```

2. Import the image from the `worldtmp.png` file into IDL:

```
worldtmpImage = READ_PNG(worldtmpFile)
```

3. Determine the size of the imported image:

```
worldtmpSize = SIZE(worldtmpImage, /DIMENSIONS)
```

4. Initialize the display objects necessary for an Object Graphics display:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [worldtmpSize[0], worldtmpSize[1]], $
   TITLE = 'Average World Temperature (in Celsius)')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0, 0, worldtmpSize[0], $
   worldtmpSize[1]])
oModel = OBJ_NEW('IDLgrModel')
```

5. Initialize the palette object, load the Rainbow18 color table into the palette, and then apply the palette to an image object:

```
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LoadCT, 38
oImage = OBJ_NEW('IDLgrImage', worldtmpImage, $
   PALETTE = oPalette)
```

6. Add the image to the model, then add the model to the view, and finally draw the view in the window:

```
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView
```

The following figure is displayed.

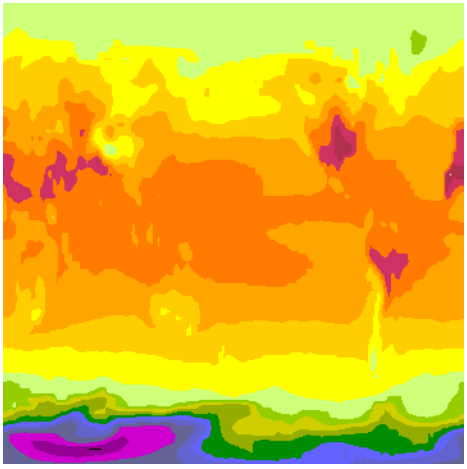


Figure 9-5: Temperature Image and Rainbow18 Color Table (Object Graphics)

Before applying the color polygons and text of each level, you must first initialize their color values and their locations. The Rainbow18 color table has

only 18 different color levels, but still has 256 elements. You can use the INDGEN routine to make an array of 18 elements ranging from 0 to 17 in value, where each element contains the index of that element. Then you can use the BYTSCL routine to scale these values to range from 0 to 255. The resulting array contains the initial color value (from 0 to 255) of the associated range (from 0 to 17, equalling 18 elements).

7. Initialize the color level parameter:

```
fillColor = BYTSCL(INDGEN(18))
```

8. Initialize the average temperature of each level, which directly depends on the initial color value of each range. Temperature is linearly scaled to range from -60 to 40 Celsius. You can convert the resulting temperature value to a string variable to be used as text:

```
temperature = STRTRIM(FIX(((20.*fillColor)/51.) - 60), 2)
```

Note

When the *fillColor* variable in the previous statement is multiplied by the floating-point value of 20 (denoted by the decimal after the number), the elements of the array are converted from byte values to floating-point values. These elements are then converted to integer values with the FIX routine so the decimal part will not be displayed. The STRTRIM routine converts the integer values to string values to be displayed as text. The second argument to STRTRIM is set to 2 to note the leading and trailing black values should be trimmed away when the integer values are converted to string values.

With the polygon color and text now defined, you can determine their locations. You can use a polygon object to draw each polygon and text objects to display each element of text. The process is repetitive from level to level, so a FOR/DO loop is used to display the entire colorbar. Since each polygon and text is drawn individually within the loop, you only need to determine the location of a single polygon and an array of offsets for each step in the loop. The following two steps describe this process.

9. Initialize the polygon and the text location parameters. Each polygon is 35 pixels in width and 18 pixels in height. The offset will move the y-location 18 pixels every time a new polygon is displayed:

```
x = [5., 40., 40., 5., 5.]
y = [5., 5., 23., 23., 5.] + 5.
offset = 18.*FINDGEN(19) + 5.
```

10. Initialize the polygon and text objects:

```
oPolygon = OBJARR(18)
oText = OBJARR(18)
FOR i = 0, (N_ELEMENTS(oPolygon) - 1) DO BEGIN & $
   oPolygon[i] = OBJ_NEW('IDLgrPolygon', x, $
   y + offset[i], COLOR = fillColor[i], $
   PALETTE = oPalette) & $
   oText[i] = OBJ_NEW('IDLgrText', temperature[i], $
   LOCATIONS = [x[0] + 3., y[0] + offset[i] + 3.], $
   COLOR = 255*(fillColor[i] LT 255), $
   PALETTE = oPalette) & $
ENDFOR
```

Note

The & after BEGIN and the $ allow you to use the FOR/DO loop at the IDL command line. These & and $ symbols are not required when the FOR/DO loop in placed in an IDL program as shown in `ApplyColorbar_Indexed_Object.pro` in the `examples/doc/objects` subdirectory of the IDL installation.

11. Add the polygons and text to the model, then add the model to the view, and finally redraw the view in the window:

```
oModel -> Add, oPolygon
oModel -> Add, oText
oWindow -> Draw, oView
```

The following figure displays the colorbar annotation applied to the image.

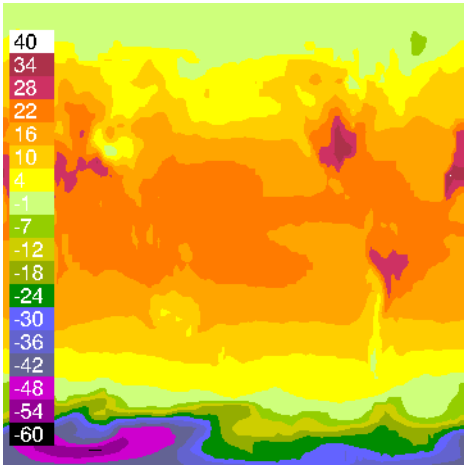


Figure 9-6: Temperature Image and Colorbar (Object Graphics

12. Clean up object references. When working with objects always remember to clean up any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view and the palette objects:

```
OBJ_DESTROY, [oView, oPalette]
```

Annotating RGB Image Objects

When using Object Graphics, colors can be defined just by the values of their red, green, and blue components. In this example, a color spectrum of additive and subtractive primary colors will be drawn on an RGB image for comparison with the colors in that image. The `glowing_gas.jpg` file (which is provided by the Hubble Heritage Team, made up of AURA, STScI, and NASA) contains an RGB image of an expanding shell of glowing gas surrounding a hot, massive star in our Milky Way Galaxy. This image contains all the colors of this spectrum. Complete the following steps for a detailed description of the process.

Example Code

See `applycolorbar_rgb_object.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for code that duplicates this example.

Run the example procedure by entering `applycolorbar_rgb_object` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT applycolorbar_rgb_object.pro`.

---

1. Determine the path to the `glowing_gas.jpg` file:

```
cosmicFile = FILEPATH('glowing_gas.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Import the image from the `glowing_gas.jpg` file into IDL:

```
READ_JPEG, cosmicFile, cosmicImage
```

3. Determine the size of the imported image. The image contained within this file is pixel-interleaved (the color information is contained within the first dimension). You can use the SIZE routine to determine the other dimensions of this image:

```
cosmicSize = SIZE(cosmicImage, /DIMENSIONS)
```

4. Initialize the display objects required for an Object Graphics display:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [cosmicSize[1], cosmicSize[2]], $
   TITLE = 'glowing_gas.jpeg')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., cosmicSize[1], $
   cosmicSize[2]])
oModel = OBJ_NEW('IDLgrModel')
```

5. Initialize the image object. The INTERLEAVE keyword is set to 0 because the RGB image is pixel-interleaved:

```
oImage = OBJ_NEW('IDLgrImage', cosmicImage, $
   INTERLEAVE = 0, DIMENSIONS = [cosmicSize[1], $
   cosmicSize[2]])
```

6. Add the image to the model, then add the model to the view, and finally draw the view in the window:

```
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView
```

   The following image contains all of the colors of the additive and subtractive primary spectrum. A colorbar annotation can be added to compare the colors of that spectrum and the colors within the image. The color of each box is defined in the following array.

Figure 9-7: Cosmic RGB Image (Object Graphics)

You can use the following to determine the color and location parameters for each polygon.

7. Initialize the color parameters:

```
fillColor = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white
```

8. After defining the polygon colors, you can determine their locations. Initialize polygon location parameters:

```
x = [5., 25., 25., 5., 5.]
y = [5., 5., 25., 25., 5.] + 5.
offset = 20.*FINDGEN(9) + 5.
```

The *x* and *y* variables pertain to the x and y locations (in pixel units) of each box of color. The *offset* maintains the spacing (in pixel units) of each box. Since the image is made up of mostly a black background, the x border of the colorbar is also determined to draw a white border around the polygons.

9. Initialize location of colorbar border:

```
x_border = [x[0] + offset[0], x[1] + offset[7], $
   x[2] + offset[7], x[3] + offset[0], x[4] + offset[0]]
```

   The y border is already defined by the y variable.

   These parameters are used when initializing the polygon and polyline objects These objects will be used draw the boxes of the color spectrum and the colorbar border. Each polygon is 20 pixels wide and 20 pixels high. The offset will move the y-location 20 pixels every time a new polygon is displayed.

10. Initialize the polygon objects. The process is repetitive from level to level, so a FOR/DO loop will be used to display the entire colorbar. Since each polygon is drawn individually within the loop, you only need to determine the location of a single polygon and an array of offsets for each step in the loop:

```
oPolygon = OBJARR(8)
FOR i = 0, (N_ELEMENTS(oPolygon) - 1) DO oPolygon[i] = $
   OBJ_NEW('IDLgrPolygon', x + offset[i], y, $
   COLOR = fillColor[*, i])
```

11. The colorbar border is produced with a polyline object. This polyline object requires a *z* variable to define it slightly above the polygons and image. The z variable is required to place the polyline in front of the polygons. Initialize the polyline (border) object:

```
z = [0.001, 0.001, 0.001, 0.001, 0.001]
oPolyline = OBJ_NEW('IDLgrPolyline', x_border, y, z, $
   COLOR = [255, 255, 255])
```

12. The polygon and polyline objects can now be added to the model and then displayed (re-drawn) in the window. Add the polygons and polyline to the model, then add the model to the view, and finally redraw the view in the window:

```
oModel -> Add, oPolygon
oModel -> Add, oPolyline
oWindow -> Draw, oView
```

   The following figure shows the colorbar annotation applied to the image.

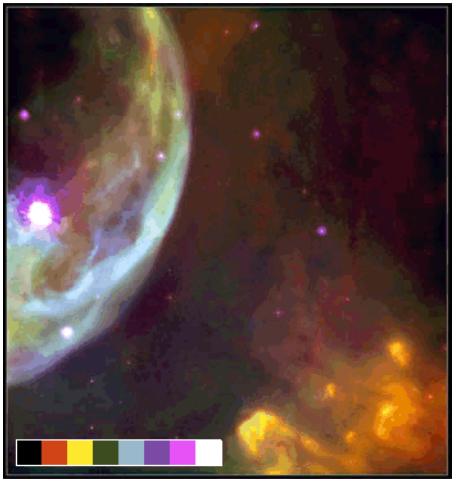
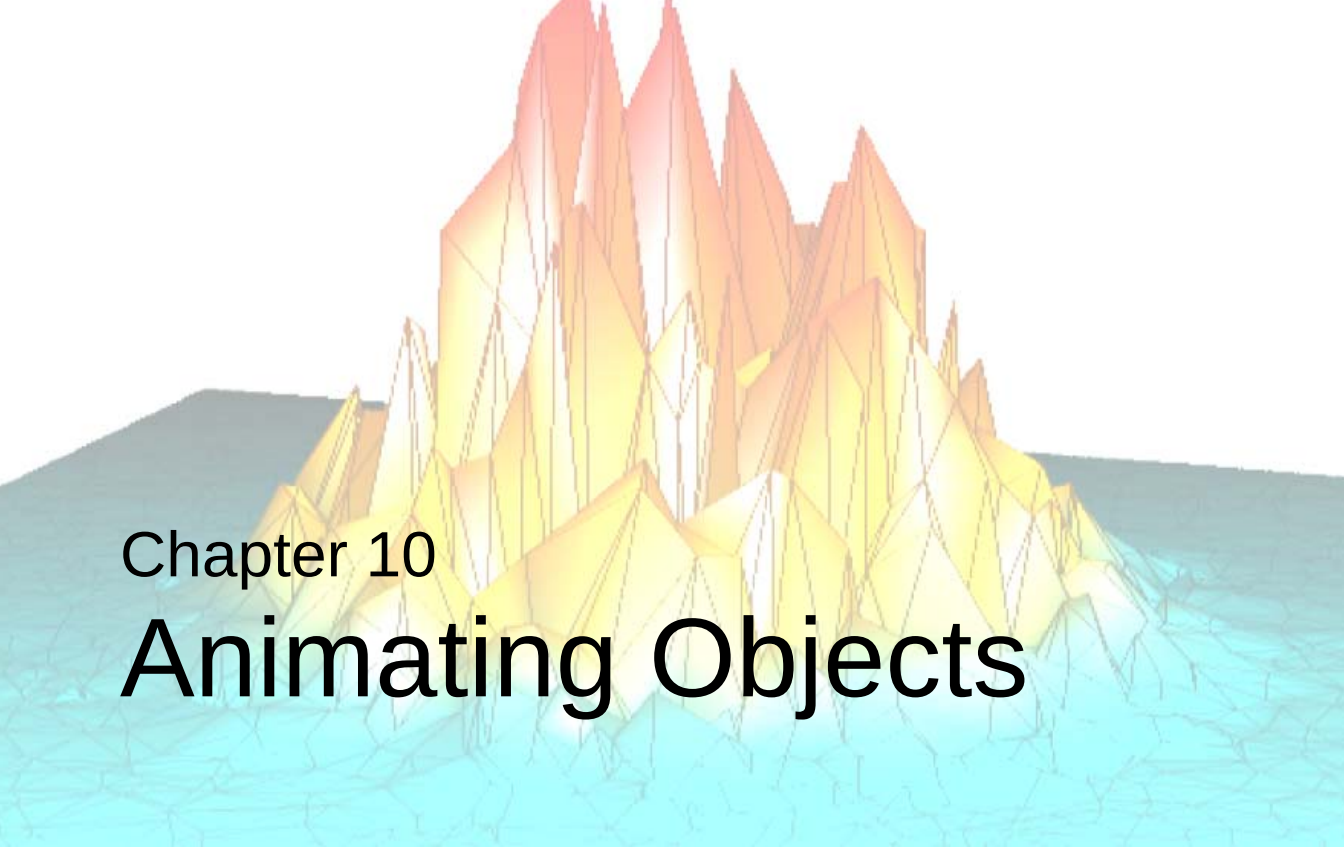


Figure 9-8: Specified Colors on an RGB Image (Object Graphics)

13. Clean up object references. When working with objects always remember to clean up any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object:

```
OBJ_DESTROY, oView
```



Chapter 10
Animating Objects

The following topics are covered in this chapter:

Overview of Object Animation

The animation functionality in IDL lets you draw a series of images in rapid succession, the speed of which has no limit other than that of system capabilities and graphics hardware. You can easily control the rate and order of the image display, or synchronize several displays. You can also display overlays that contain other types of information (such as text, ROIs, or contours) that are either specific to the currently displayed image or common to all displayed images. In addition to images, other objects such as surface and volume objects can also be animated.

The key to this flexibility is due to the fact that animation capabilities are provided in part by an IDLgrModel object, to which you can add any combination of graphic objects. As shown in the following figure, the object hierarchy for animation is very similar to a standard window-scene-view-model hierarchy of a typical display. When the animation model is used in conjunction with an IDLitWindow and a custom behavior object, the animation display possibilities are nearly limitless.

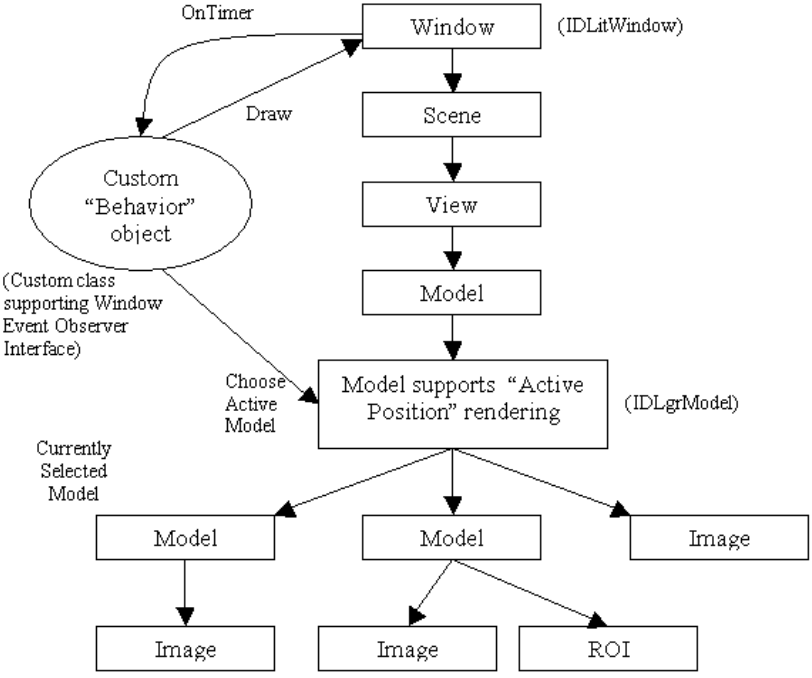


Figure 10-1: Object Interaction in Animation Support

While the graphics tree of an animation display is very similar to a standard display, it is important to note the differences. Animation relies on an IDLitWindow (not IDLgrWindow), which has a built-in timer mechanism, and an IDLgrModel object that has an awareness of "Active Position" rendering. There is also a user-defined object that determines how the contents in the animation model are modified. This *behavior object* can incorporate any action, but it will commonly iterate through a series of images or transform a model object in response to a timer signal received from the window. See the following topics for more information on adding animation functionality to a program or application:

- "Configuring an Animation Model Object" on page 248 — describes how IDLgrModel properties enable animation
- "Controlling the Animation Rate" on page 250 — describes how to incorporate behavior objects, and how to set IDLgrWindow methods and parameters to start, stop, and control the rate of an animation
- "Designing a Behavior Object" on page 251 — describes the most important elements of a behavior object, and provides access to two working animation examples (a simple Cine loop, and a timer-based surface rotation)

For an example that incorporates animation elements into a widget application that lets you interactively control the playback of a series of image frames, see "Example: Interactive Cine Animation" on page 255.

Note

For information on how scene contents, image sizes, and display refresh rates influence animation performance, see "Factors Affecting Animation Performance" on page 253.

Configuring an Animation Model Object

An IDLgrModel object that supports animation acts as a container for any number of objects. However, instead of displaying all objects when the model is drawn, a model object that supports animation lets you instruct the object to draw only one of the objects in its container. For example, this allows you to display a succession of single images from a series of images that has been added to the animation model.

To create a model object that supports animation, set the RENDER_METHOD property value to 1 to display single objects from the model collection. Use the ACTIVE_POSITION property of the model, a zero-based index into the model collection, to define what object to display. The default RENDER_METHOD value (0), draws all objects in a model. If your animation model contains a single object (e.g., when you are rotating a surface), you do not need to set the RENDER_METHOD property.

The index of the item to draw does not automatically increment when the model is drawn, so redrawing the scene graph always draws the same content. This maintains the window contents when the window is refreshed or resized. Therefore, the model object must be explicitly told which item to draw. The logic that determines which item to draw is left to the application and is typically encapsulated in a user-defined behavior class. See “Designing a Behavior Object” on page 251 for more information.

Using Multiple Models

It is suggested that you create a main-level display model (that renders in the traditional all-object fashion) in addition to the animation model (that sets the RENDER_METHOD property). This compartmentalization provides more flexibility in terms of display content. For example, suppose you have a Cine display. The main-level display model could contain a text object that is displayed on all image frames. If you did not have the main-level model, the text would only appear as part of the Cine, according to its position in the animation model.

If you are adding more than just images to an animation model (e.g., you want a contour or ROI overlaying an image), then you can create additional sub-models. These are useful when each frame of an animation is a composite of several

individual, data-specific objects. The following figure provides a simple illustration of a possible model hierarchy in animation.

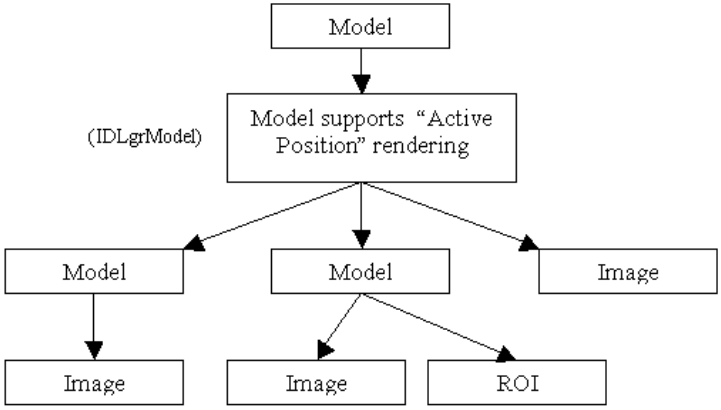


Figure 10-2: Possible Model Object Hierarchy in an Animation Display

Typically, images can be added directly to a model object. However, if your application provides a way to interactively change the properties of the images (e.g., by filtering or modifying the color table), you should add the images to an object collection. You can then pass a pointer to this object array, and access the images when needed. This is significantly easier than accessing the image data from the animation model. The following short segment of code shows such an image collection, `oImageColl`, and the animation model, `oAnimationModel`:

```
; Access the image data.
head = READ_BINARY( FILEPATH('head.dat', $
        SUBDIRECTORY=['examples','data']), $
        DATA_DIMS=[80,100, 57])

; Initialize an object array with dimensions equal to the
; number of images in the series.
oImageColl = OBJARR(57)

; Create the image objects, add each to the image collection and
; the animation model.
FOR i=0, 56 DO BEGIN
   oImageColl[i] = OBJ_NEW('IDLgrImage', head[*,*,i],
      PALETTE=oPalette, /INTERP)
   oAnimationModel->Add, oImageColl[i]
ENDFOR
```

Controlling the Animation Rate

A custom behavior object typically controls the display of objects in a model that supports animation. However, it is the IDLitWindow object that controls the timing of the animation, and notifies the behavior object that it is time to initiate an action. To define what behaviors are initiated when a timer event occurs, add one or more behavior objects to the AddWindowEventObserver method observer list.

To enable timer events for a window, you need to use the SetEventMask method. This effectively lets you to turn on or turn off an animation by enabling or disabling a window's ability to respond to timer events. (Use the GetEventMask method to determine which events are enabled in a window.)

The SetTimerInterval method determines the animation rate. Use the SetTimerInterval method to set a value that specifies how many seconds pass before the next timer event occurs. In the following sample code, `oAnimBehavior` is the custom behavior object, `oAnimationModel` is the model that contains the animation, and `oWin` is an IDLitWindow object.

```
; Create a custom animation object and initialize it with
; the animation model. Add the new object to the list
; of window observers and set the display rate (10 frames
; per second).
oAnimBehavior = OBJ_NEW('MyAnimation', oAnimationModel)
oWin->AddWindowEventObserver, oAnimBehavior
oWin->SetTimerInterval, 0.1

; Play the animation.
oWin->SetEventMask, /TIMER_EVENTS
```

To turn off an animation, set TIMER_EVENTS equal to 0.

The SetTimerInterval method interval value determines how often an IDLitWindow object calls the OnTimer method for the behavior objects in the observer list. Therefore, each animation behavior object must implement the OnTimer method. See "Designing a Behavior Object" on page 251 for more information.

Designing a Behavior Object

A behavior object is an instance of a custom class that controls the display of the object(s) contained in a model object that supports animation. This behavior object determines what action to take in response to a timer event. When a timer event occurs, the window object calls the OnTimer method of each window observer (each behavior object) that implements it. The following figure shows the interaction between the window and a behavior object.

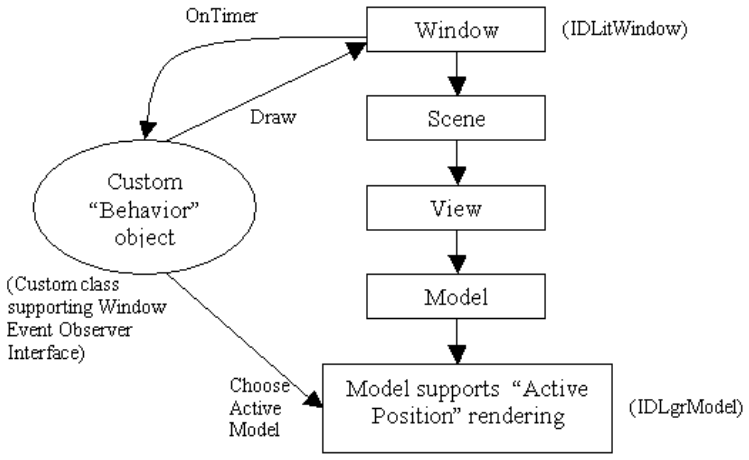


Figure 10-3: Interaction Between Window, Behavior Object, and Animation Model

In the example of a Cine, the behavior object's OnTimer method tells the model object which model or graphic to display the next time the scene is drawn. The behavior object completes its action by signaling the window object to draw the scene with the updated model. The system is quiescent until the next timer interval expires, at which point the process begins again. In widget applications, widget events and other application processing may occur during the quiet time.

The OnTimer method of the behavior object need not be complex. The following simple OnTimer method of the user-defined behavior object, `MyAnimation`, simply iterates through the frames in an image series. The OnTimer method parameter specifies the IDLitWindow object in which the timer event occurred.

```
PRO MyAnimation::OnTimer, oWin

; Add one to the current frame number.
self.currentFrame++

; Iterate through the image frames. Define the frame to display
; by setting the ACTIVE_POSTION property on the model.
IF self.currentFrame GE self.oAnimationModel->Count() THEN $
   self.currentFrame = 0
self.oAnimationModel->SetProperty, $
   ACTIVE_POSITION=self.currentFrame

; Draw the scene.
oWin->Draw

END
```

Example Code

For the simple Cine animation example, see `animation_image_doc.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory. Run the example procedure by entering `animation_image_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT animation_image_doc.pro`. This example shows a simple animation in a window that continues until the window is closed.

More than one behavior object can be associated with a window, which lets you create synchronous animations. The window triggers all behaviors associated with it by calling all observers that are interested in OnTimer notifications. Also, a behavior can be programmed to perform any arbitrary operation. It is not limited to cycling through a series of images. For example, it could alter a transform in a model object to implement a time-based rotation.

Example Code

For a simple surface rotation animation example, see `animation_surface_doc.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory. Run the example procedure by entering `animation_surface_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT animation_surface_doc.pro`.

Factors Affecting Animation Performance

Animation performance depends on a large number of factors that include the amount of graphic content in each frame and the capabilities of the hardware. You adjust the animation rate by setting the timer interval value of the IDLitWindow object. When the timer interval expires, IDL calls the OnTimer method of the behavior objects that are observing the window. If the hardware can draw the entire scene graph within the requested timer interval, IDL waits until the timer interval expires before calling the OnTimer methods again, in order to produce the requested animation rate. If IDL was performing another operation or computation when the timer interval began, it returns to that task after drawing the scene and until the time interval expires again. If IDL cannot draw the entire scene graph before the timer interval expires, it finishes drawing the scene graph and immediately moves on to the next frame by calling the OnTimer method again, as long as the window timer is running. Any excess timer expirations are discarded so they do not “pile up” behind the animation. Therefore, you may experience a “maximum possible frame rate” that depends on the graphic content and the capabilities of the machine you are using.

Scene graphs that contain a large amount of graphical information and/or render slowly can reduce the maximum achievable frame rate. Very large polygonal meshes and volumes are examples of graphical content that will reduce animation performance.

Multiple Image Copies

If you are animating a very large amount of image data, the maximum frame rate may also be reduced if the total amount of image data exceeds the space available on the video card and system memory. IDL attempts to optimize image rendering by keeping image data in the video card memory and in system memory as video card memory is exhausted. If the image memory requirements exceed the amount of space available in “fast” memory, (video and system memory), the system may move image data out to “slow” memory (paging space). This can reduce image animation performance as older images need to be swapped back into video memory when they need to be displayed again. If this occurs, consider using a single IDLgrImage object in your animation and replace the image data in the image object with image data for the next frame in the OnTimer method. This reduces the total number of copies of image data stored in memory at once and still provides good performance. It is best to put all your image data into IDLgrImage objects when the images all fit into memory and there is a requirement to rapidly animate all the images in a loop. If all the images do not fit into memory or if rapid access to all the images is not necessary, it may be better to use a single IDLgrImage object.

Graphics Display Refresh Rate

Maximum frame rates may also be restricted by the refresh rate of your graphics display device if the screen refresh rate is tied to applications. This can prevent the application from exceeding the refresh rate of the display device, which is often in the range of 60-120 frames per second. If you find that you cannot create an animation faster than the refresh rate, look for a setting on your video card control software to disable this synchronization. It is often referred to as VSYNC, vertical synchronization, or “refresh rate override”.

Using application frame rates in excess of display device frame rates with synchronization turned off is often not useful and can even be distracting because of missing or “dropped” frames. For example, if you try to display a 10-image animation on a display device using a 60 Hz refresh rate at 600 frames per second, the animation will appear stalled, since the user will see the same image over and over. The other 9 images are drawn to the display between display device refreshes and are “dropped”.

Example: Interactive Cine Animation

You can incorporate animation into a widget application by using the CLASSNAME keyword to assign an IDLitWindow object to WIDGET_DRAW and using the properties and methods documented in this chapter. The following widget application lets you start and stop an animation, and set the frame rate and frame increment. It is limited to this functionality only to highlight the essential features of animation. You could incorporate zooming, panning, or the addition of annotative objects (such as text, ROIs, or contours) in either the main-level model or in individual object models. See “Using Multiple Models” on page 248 for information on how the placement of objects and models in the graphics hierarchy affects the display.

Example Code

See `animation_doc.pro` in the `examples/doc/objects` subdirectory of the IDL installation directory for the complete widget animation example. Run the example procedure by entering `animation_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT animation_doc.pro`.

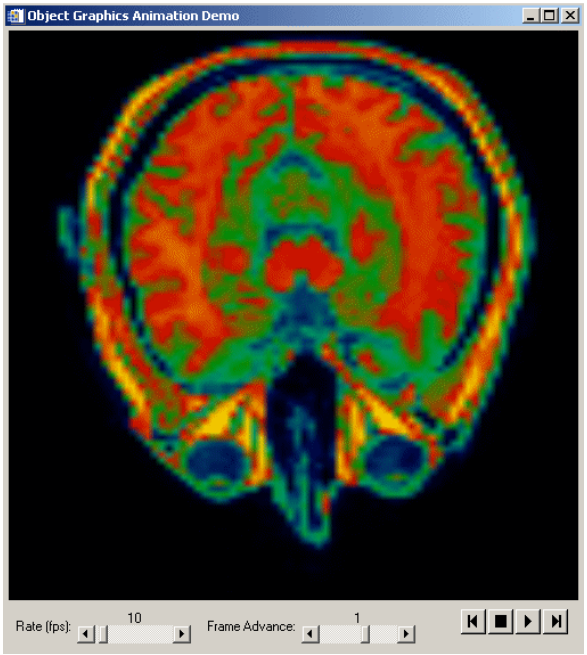
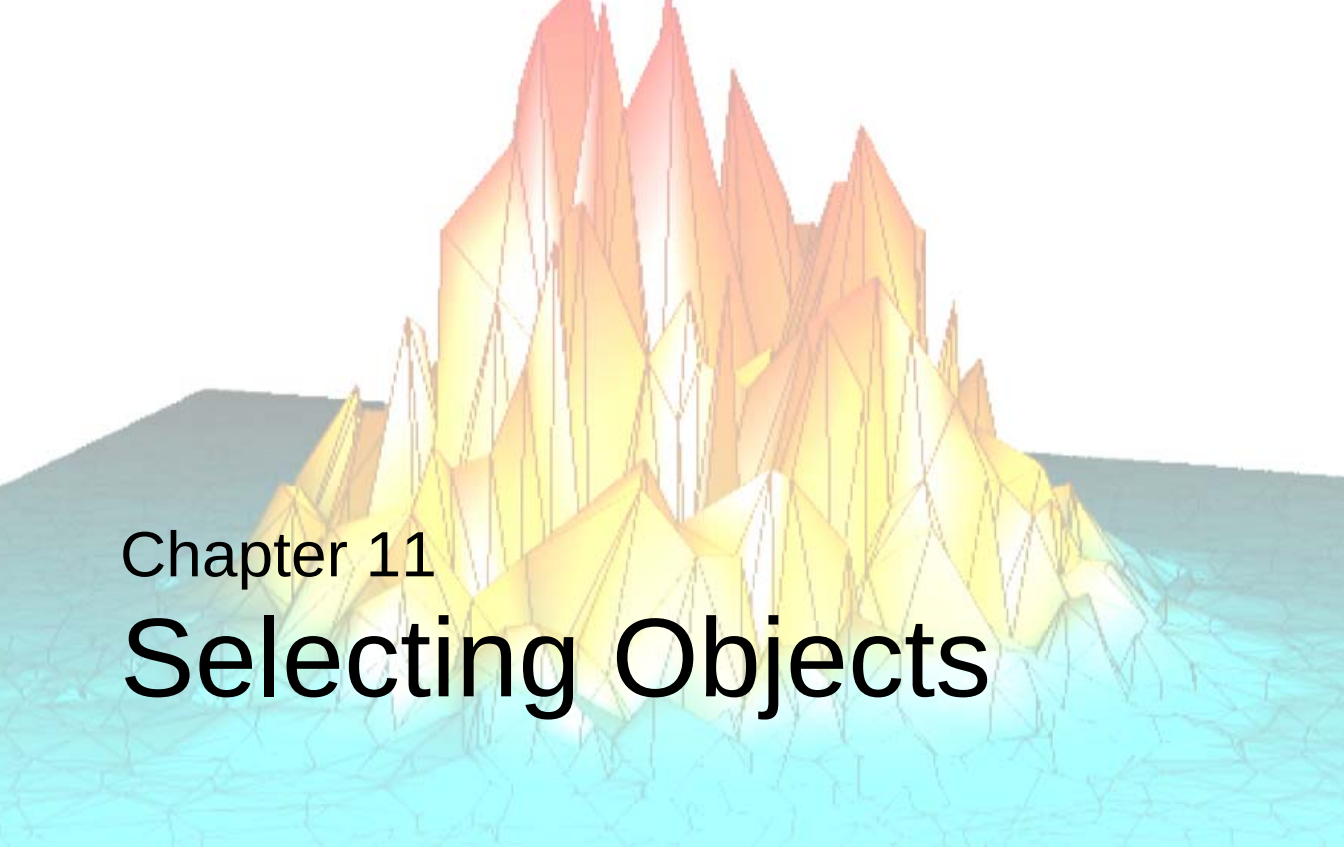


Figure 10-4: Simple Widget Animation Interface



Chapter 11
Selecting Objects

This chapter will describe the IDL Object Graphics selection and direct manipulation features. The following topics are covered in this chapter:

Selection and Data Picking

When graphical items are drawn to a window, it is often useful to be able to click the mouse on a certain location and request a list of the items that are displayed at that particular location. In IDL, this is called selection. Because IDL object graphics are retained in memory, they can be uniquely identified by their individual object references, and therefore can be reported as having been selected.

In many cases, it is also useful to be able to request the data value of the object at the user-selected location. In IDL, this is called data picking.

Object Selection

With object graphics, the process of selection is very similar to drawing, except that nothing is displayed on the screen, and information about which objects were selected is returned to the user. Selection is performed via the Select method of an IDLgrWindow object.

Three types of objects may be selected: view objects, model objects, and visualization objects. For a given scene that contains more than one view, you can use the Select method to determine which view is selected at a given location. Likewise, for a given view, you can use the Select method to determine which models and/or visualization objects within that view are selected.

An object is considered to be selected if its graphical rendering falls within a box centered on a given location. The dimensions of the box are set via the DIMENSIONS keyword to the Select method. Both the location argument and dimensions keyword values are measured in units specified via the UNITS keyword.

The Select method returns a vector of objects, sorted in depth order (nearest to the eye is first), that meet the criteria of having been selected at the given location. If no objects are selected at the given location, the Select method returns –1.

See "IDLgrWindow::Select" (*IDL Reference Guide*) for a detailed description of the Select method.

Selecting Views

To determine which of a set of views within a given scene are selected at a given location, call the Select method on an IDLgrWindow object with an instance of an IDLgrScene object as its first argument, and the location at which the selection is to occur as its second argument:

```
myLoc = [myMouseEvent.x, myMouseEvent.y]
mySelectedViews = myWindow->Select(myScene, myLoc)
```

Selecting Visualization Objects

To determine which visualization objects within a given view are selected at a given location, call the Select method on an IDLgrWindow object with an instance of an IDLgrView object as its first argument, and the location at which the selection is to occur as the second argument:

```
myLoc = [myMouseEvent.x, myMouseEvent.y]
mySelectedGraphics = myWindow->Select(myView, myLoc)
```

Note

If a model within the view is set as a selection target, the model object, rather than its contained visualization objects, is returned in the vector of selected objects.

Selecting Models

In some cases, a group of visualization objects may be considered subcomponents of the model in which they are contained. As a result, you may want to know when a model object (rather than one or more of its contained visualization objects) has been selected. To enable selection of a model (rather than its visualization objects), the model object must be marked as a selection target.

To mark a model as being a selection target, set the SELECT_TARGET property of the model object to a nonzero value.

```
myWindow = OBJ_NEW('IDLgrWindow')
myView = OBJ_NEW('IDLgrView')
myModel = OBJ_NEW('IDLgrModel')
myView->Add, myModel
myModel->SetProperty, /SELECT_TARGET
myAxis = OBJ_NEW('IDLgrAxis', 0)
myModel->Add, myAxis
myWindow->Draw, myView
```

In the above example, if a selection at location [*myX*, *myY*] would normally select the axis object, the returned value of the Select method will be the object reference to myModel rather than the object reference to myAxis.

A Selection Example

An example procedure named `sel_obj.pro` creates two views, places models within the views, and provides an interface to let you choose between selecting models or visualization objects. A mouse click in one of the views will update a label that identifies the current selections.

Example Code

This example, `sel_obj.pro`, is included in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `sel_obj` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT sel_obj.pro`.

Data Picking

To get the data value that corresponds to a particular window location, use the PickData method of an IDLgrWindow object. Note that you must draw the view to the window before calling the PickData method.

```
myLoc = [myMouseEvent.x, myMouseEvent.y]
result = myWindow->PickData(myView, myModel, myLoc, returnedXYZ)
```

The PickData method returns one of the following values:

- 0 (zero) if the pick hit the background of the view
- 1 (one) if the pick hit the one of the visualization objects in the view
- –1 if an error occurred (for instance, if the pick location lies outside of the given view)

The data value at the pick is returned in the *returnedXYZ* argument. This value represents the mapping of the window location to the data space of the model.

The PickData method relies on the contents of the depth buffer at the time it is called to compute and return its results. Be sure that the depth buffer contents are appropriate for getting the expected results from PickData.

Note

If you set the DEPTH_WRITE_DISABLE or DEPTH_TEST_DISABLE property of an object to prevent an object from modifying the depth buffer as it is drawn, this also prevents the object from being located by the PickData method (the return value will be 0).

A Data Picking Example

The example procedure `surf_track.pro` includes code using the PickData method to retrieve data values from a surface object. `This example` is described in "An Interactive Surface Example" on page 189.

Example Code

See `surf_track.pro`, located in the `examples/doc/objects` subdirectory of the IDL distribution. Run the example procedure by entering `surf_track` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT surf_track.pro`.

Chapter 12
Displaying, Copying and Printing Objects

The following topics are covered in this chapter:

Overview of Object Graphic Destinations

Once a graphic object tree has been created, it can be displayed, or drawn, to a physical destination device (such as a computer screen or printer), to a memory location (such as a buffer or the operating system clipboard), or to a particular file format (such as a VRML file). Destination objects represent the final locations to which object graphics are drawn, and provide methods that allow you to control the properties of the physical device, memory buffer, or file format.

Each destination object includes a GetFontnames method, which returns the list of available fonts that can be used in IDLgrFont objects. This method will only return the names of the available TrueType fonts. Hershey fonts will not be returned as they are fixed—see Appendix H, “Fonts” (*IDL Reference Guide*) for more information.

There are five destination objects:

- buffers (IDLgrBuffer objects)
- clipboards (IDLgrClipboard objects)
- printers (IDLgrPrinter objects)
- VRML files (IDLgrVRML objects)
- windows (IDLgrWindow objects)

Of the five destination objects, Window objects are the most common and most often used, and will be addressed first.

Note

Output to IDLgrClipboard and IDLgrPrinter objects can be in bitmap or vector format. See “Bitmap and Vector Graphic Output” on page 284 for information on choosing a suitable graphics output type based on scene content.

Window Objects

Objects of the IDLgrWindow class represent a rectangular area on a computer screen into which graphics hierarchies can be rendered. Window objects can be either stand-alone windows on the screen or drawable areas in an IDL draw widget.

Creating Window Objects

There are two ways to create window objects: directly via the window object's Init method and indirectly by creating a draw widget that uses a window object as its drawable area.

Using the Init Method

The IDLgrWindow::Init method takes no arguments. Use the following statement to create a window object:

```
myWindow = OBJ_NEW('IDLgrWindow')
```

The window is displayed on the screen as soon as it has been created.

Creating a Draw Widget that Uses a Window Object

To create a draw widget that uses an Object Graphics window object rather than a Direct Graphics window for its drawable area, set the GRAPHICS_LEVEL keyword to the WIDGET_DRAW function equal to 2:

```
drawwid = WIDGET_DRAW(base, GRAPHICS_LEVEL=2)
```

Once the draw widget has been realized, you can then retrieve the object reference to the draw widget's window object using the WIDGET_CONTROL procedure:

```
WIDGET_CONTROL, drawwid, GET_VALUE=myWindow
```

Color Model

By default, window objects use the RGB color model. To create a window that uses the Indexed color model, set the COLOR_MODEL property of the window object equal to 1 (one) when creating the window:

```
myWindow = OBJ_NEW('IDLgrWindow', COLOR_MODEL=1)
```

You cannot change the color model used by a window after it has been created.

See "Color in Object Graphics" on page 46for a discussion of the two color models.

Note on Window Size Limits

The OpenGL libraries IDL uses impose limits on the maximum size of a drawable area. The limits are device-dependent — they depend both on your graphics hardware and the setting of the RENDERER property. Currently, the smallest maximum drawable area on any IDL platform is 1280-by-1024 pixels; the limit on your system may be larger.

Using Window Objects

To render a graphics tree to a window, call the IDLgrWindow::Draw method. The argument must be either an IDLgrView object or an IDLgrScene object.

```
myWindow->Draw, myView
```

or

```
myWindow->Draw, myScene
```

All objects contained within the view or scene object will be drawn to the window.

Erasing a Window

To erase the contents of a window, call the IDLgrWindow::Erase method. You can optionally supply a color to use to clear the window. By default, the window is erased to white.

For example, to erase the window to black:

```
myWindow->Erase, COLOR=[0,0,0]
```

Exposing or Hiding a Window

To expose a window so that it is the front-most window on the screen, call the IDLgrWindow::Show method with a nonzero value as the argument:

```
myWindow->Show, 1
```

To hide a window, call the IDLgrWindow::Show method with a zero value as the argument:

```
myWindow->Show, 0
```

Iconifying a Window

To iconify (minimize) a window, call the IDLgrWindow::Iconify method with a nonzero value as its argument:

```
myWindow->Iconify, 1
```

To restore an iconified window, call the IDLgrWindow::Iconify method with a zero value as its argument:

```
myWindow->iconify, 0
```

Setting the Window Cursor

To set the appearance of the mouse cursor in an IDLgrWindow object, call the IDLgrWindow::SetCurrentCursor method with a string argument representing the name of the cursor. Valid string values for the cursor name argument are:

ARROW	CROSSHAIR
ICON	IBEAM
MOVE	ORIGINAL
SIZE_NE	SIZE_NW
SIZE_SE	SIZE_SW
SIZE_NS	SIZE_EW
UP_ARROW	

The following statement sets the cursor to an up arrow:

```
myWindow->SetCurrentCursor, 'UP_ARROW'
```

The ORIGINAL cursor sets the cursor to the window system's default cursor.

See "IDLgrWindow::SetCurrentCursor" (*IDL Reference Guide*) for details on cursor values.

Saving/Restoring Windows

When an instance of an IDLgrWindow object is restored via the RESTORE procedure), it is not immediately displayed on the screen. It will be displayed as soon as one of its methods (Draw, Erase, Iconify, etc.) is called.

Saving Window Contents to a File

If you have created a scene or view containing graphical objects and wish to save the rendering to a file, you will first need to create an image object from which to retrieve the image data. The following steps render an object to a window, create an image object from the window, and save the image data as a TIFF file.

First, create the view to be rendered. Use an indexed color model for the window object, setting the background color to white and the foreground color of the plot object to black.

```
mywindow = OBJ_NEW('IDLgrWindow', COLOR_MODEL=1)
myview = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT=[0,-4,10,8], COLOR=255)
mymodel = OBJ_NEW('IDLgrModel')
myplot = OBJ_NEW('IDLgrPlot', RANDOMN(seed, 10), COLOR=0, $
   THICK=3)
; Organize the object hierarchy:
myview->Add, mymodel
mymodel->Add, myplot
; Draw to the window:
mywindow->Draw, myview
; Next, use the window object's Read method to create
; an image object with the rendered scene as its image data:
myimage = mywindow->Read()
; Retrieve the image data using the GetProperty method
; of the image object:
myimage->GetProperty, DATA=image
; Display the image data using Direct Graphics:
TV, image
; Write the image to a TIFF file named myfile.tif:
WRITE_TIFF, 'myfile.tif', image
```

Improving Window Drawing Performance

The following sections describe how to optimize drawing performance in your object graphics programs. See “Performance Tuning Object Graphics” in Chapter 2 for general notes on rendering performance.

Retained Graphics and Expose Events

During the course of an IDL session, it is possible that an IDL window will be obscured by another window. When the hidden window is brought to the front, its contents need to be regenerated. The user interface toolkit portions of the window are repaired automatically. However, the drawable portion of the window (in which graphics are rendered) requires special attention. The user can choose between two methods to handle this situation. The first option is to set the RETAIN property on the IDLgrWindow object to 2, which suggests that IDL is required to retain a backing store of the entire contents of the window. When the window is exposed, the backing store will be copied to the screen. The second option is to set the RETAIN property to 0 (no retention), and to request that expose events are to be reported for draw widgets. Whenever a portion of the window becomes exposed, an event is generated. The event handler for the drawable can then re-issue a draw of the appropriate contents for that window.

While the second option may seem a bit more complicated, it is to the users advantage to take this approach for performance reasons. When RETAIN is 0, the window device drivers are able to utilize a double-buffered rendering scheme that can capitalize on hardware acceleration. For interactive applications, this hardware acceleration can have a crucial impact on the perceived manipulation capabilities of the interface. When RETAIN is 2, on the other hand, IDL will render to an off screen pixmap, which often relies on a software implementation. If several drawing calls are issued in a row, the performance may be noticeably slower.

Instancing to Improve Redraw Performance

Within interactive graphics applications, it is often necessary to redraw a given view over and over again (for example, as the user clicks and drags within the view to manipulate one or more objects). During those redraws, it may be that only a small subset of the objects within the view is changing, while the remaining objects are static. In such a case, it may be more efficient to take a snapshot of the unchanged portion of the view. This snapshot can be reused for each draw, and only the changing portion of the view needs to be re-rendered. This process is called instancing.

It is to your advantage to use instancing only in cases where displaying the snapshot image is faster than rendering each of the objects that remain unchanged.

The following example shows how a typical instancing loop would be set up. First, hide the objects in the view that will be changing. In this example, we assume that the objects that change continuously are contained by a single model object, with the object reference myChangingModel. We set the HIDE property for this model to remove it from the rendered view.

```
myChangingModel->SetProperty, HIDE=1

;Next, create an instance of the remaining portion
;of the view by setting the CREATE_INSTANCE keyword to
;the window's Draw method:
myWindow->Draw, myView, /CREATE_INSTANCE

;Next, hide the unchanging objects.
;Assume that the unchanging portion of the
;scene is contained in a single model object.
myUnchangingModel->SetProperty, HIDE=1

;Set the HIDE property for the changing model
;object equal to zero, revealing the object:
myChangingModel->SetProperty, HIDE=0

;Set the view object's TRANSPARENT property.
;This ensures that we will not erase the
;instance data (the unchanging part of the scene)
;when drawing the changing model.
myView->SetProperty, /TRANSPARENT

;Next, we set up a drawing loop that will render
;the changing model. For example, this loop might
;rotate the changing model in 1 degree increments.
ROT = 0
FOR i=0,359 DO BEGIN
   ROT=ROT+1
   myChangingModel->Rotate, [0,1,0], ROT
   myWindow->Draw, myView, /DRAW_INSTANCE
ENDFOR

;After the drawing loop is done, ensure nothing is hidden,
;and that the view erases as it did before:
myUnchangingModel->SetProperty, HIDE=0
myView->SetProperty, TRANSPARENT=0
```

Buffer Objects

Objects of the IDLgrBuffer class represent a memory buffer into which graphics hierarchies can be rendered. Object trees can be drawn to instances of the IDLgrBuffer object and the resulting image can be retrieved from the buffer using the Read() method. The off-screen representation avoids dithering artifacts by providing a full-resolution buffer for objects using either the RGB or Color Index color models.

Creating Buffer Objects

The IDLgrBuffer::Init method takes no arguments. Use the following statement to create a buffer object:

```
myBuffer = OBJ_NEW('IDLgrBuffer')
```

This creates an object that is available as a destination device to be rendered into or copied from.

See “IDLgrBuffer” (*IDL Reference Guide*) for details on creating and using buffer objects.

Clipboard Objects

Objects of the IDLgrClipboard class send Object Graphics output to the operating system native clipboard or to a file in bitmap or vector format. The file type and destination is dependent upon the platform and the values of Draw method keywords.

Note

What appears when producing bitmap or vector output is dependent upon several factors. See "Bitmap and Vector Graphic Output" on page 284 for details.

Writing to a File from IDLgrClipboard

The file type produced when the IDLgrClipboard::Draw method is passed an IDLgrView, IDLgrViewgroup, or IDLgrScene object varies depending upon keyword settings and the platform on which the call is issued. If the FILENAME keyword is set to a non-empty string, the name of the file IDL creates is specified by the string. If the FILENAME keyword is a non-zero, numeric value, IDL creates a file named `idl.`*`ext`* where *`ext`* is replaced with the appropriate extension shown in parentheses in the following table.

Keyword Settings	Windows File Type	UNIX File Type
VECTOR = 1, POSTSCRIPT = 1	Encapsulated PostScript (EPS)	Encapsulated PostScript (EPS)
VECTOR = 1, POSTSCRIPT = 0	Enhanced MetaFile (EMF)	Encapsulated PostScript (EPS)
VECTOR = 0, POSTSCRIPT = 1	Encapsulated PostScript (EPS)	Encapsulated PostScript (EPS)
VECTOR = 0, POSTSCRIPT = 0	Bitmap (BMP)	Encapsulated PostScript (EPS)

Table 12-1: File Types Produced by IDLgrClipboard Draw Method

Note

PostScript clipboard output can be generated using the CMYK color model. See the IDLgrClipboard::Draw method in the *IDL Reference Guide* for details.

Writing to the Clipboard from IDLgrClipboard

Objects can be written to the operating system clipboard using IDLgrClipboard::Draw. When the FILENAME keyword equals an empty string (" "), equals 0 (zero), or is not specified, the output is written to the clipboard.

Note

The IDLgrClipboard object empties the Windows clipboard before writing to it.

Creating Clipboard Objects

The IDLgrClipboard::Init method takes no arguments. Use the following statement to create a clipboard object that represents the system-native clipboard buffer:

```
myClipboard = OBJ_NEW('IDLgrClipboard')
```

The following code creates an IDLgrClipboard object and outputs the contents of an IDLgrView, IDLgrViewgroup, or IDLgrScene to various files based on the platform. This is useful to determine exactly how the contents of the window are translated into bitmap or vector graphics. In the following code, `myview` denotes the name of the object (view, viewgroup, or scene) to be output. Vector postscript output is also generated using the CMYK color model.

```
oClip = OBJ_NEW('IDLgrClipboard')

; Create Windows-only output file types.
if !VERSION.OS_FAMILY eq 'Windows' then begin
   oClip->Draw, myview, VECTOR=0, POSTSCRIPT=0, $
      FILENAME="clipboard.bmp"
   oClip->Draw, myview, VECTOR=1, POSTSCRIPT=0, $
      FILENAME="clipboard.emf"
endif

; Create bitmap and vector PostScript files.
oClip->Draw, myview, VECTOR=0, POSTSCRIPT=1, $
   FILENAME="clipboard_bitmap.eps"
oClip->Draw, myview, VECTOR=1, POSTSCRIPT=1, $
   FILENAME="clipboard_vector.eps"

   oClip->Draw, myview, VECTOR=1, POSTSCRIPT=1, $
      /CMYK, FILENAME="clipboard_cmyk.eps"

obj_destroy, oClip
```

See “IDLgrClipboard” (*IDL Reference Guide*) for details.

Printer Objects

Objects of the IDLgrPrinter class represent a physical printer onto which graphics hierarchies can be rendered in either bitmap or vector mode. What appears when producing bitmap or vector output depends upon several factors. See “Bitmap and Vector Graphic Output” on page 284 for details.

Creating Printer Objects

The IDLgrPrinter::Init method takes no arguments. Use the following statement to create a printer object:

```
myPrinter = OBJ_NEW('IDLgrPrinter')
```

This creates an object that maintains information about the printer. By default, this information pertains to the default printer installed for your system. To select a different printer or setup attributes of the printer, use the printer dialogs described in the next section.

See “IDLgrPrinter” (*IDL Reference Guide*) for details on creating printer objects.

Color Model

By default, printer objects use the RGB color model. To create a printer that uses the Indexed color model, set the COLOR_MODEL property of the printer object equal to 1 (one) when creating the printer:

```
myWindow = OBJ_NEW('IDLgrPrinter', COLOR_MODEL=1)
```

You cannot change the color model used by a printer after it has been created.

See “Color in Object Graphics” on page 46 for a discussion of the two color models.

Printer Dialogs

IDL includes two functions useful for controlling printers and print jobs.

DIALOG_PRINTERSETUP

Call the DIALOG_PRINTERSETUP function with the object reference of a printer object as its argument to open an operating system native dialog for setting the applicable properties of a particular printer. DIALOG_PRINTERSETUP returns a nonzero value if you pressed the **OK** button in the dialog, or zero otherwise.

```
result = DIALOG_PRINTERSETUP(myPrinter)
```

See DIALOG_PRINTERSETUP in the *IDL Reference Guide* for details.

DIALOG_PRINTJOB

Call the DIALOG_PRINTJOB function with the object reference of a printer object as its argument to open an operating system native dialog to initiate a printing job. DIALOG_PRINTJOB returns a nonzero value if you pressed the **OK** button in the dialog, or zero otherwise.

```
result = DIALOG_PRINTJOB(myPrinter)
```

See DIALOG_PRINTJOB in the *IDL Reference Guide* for details.

Drawing to a Printer

To draw a graphics tree to a printer, call the IDLgrPrinter::Draw method. The argument must be either an IDLgrView object, an IDLgrViewGroup object, or an IDLgrScene object.

```
myPrinter->Draw, myView
```

or

```
myPrinter->Draw, myScene
```

All objects contained within the scene, viewgroup, or view will be drawn to the printer.

Note

The scene or view to be drawn may be the same as the scene or view being displayed in one or more windows.

Printing in Bitmap or Vector Graphic Mode

The IDLgrPrinter::Draw method VECTOR keyword specifies whether the output is in bitmap or vector format. The following table shows the keyword options and results for each platform.

Keyword Settings	Windows Printer Output	UNIX File Type
VECTOR = 0	Bitmap (BMP)	Encapsulated PostScript (EPS) file (e.g. `xprinter.eps`)
VECTOR = 1	Enhanced MetaFile (EMF)	Encapsulated PostScript (EPS) file (e.g. `xprinter.eps`)

Table 12-2: File Types Produced by IDLgrPrinter Draw Method

VECTOR=0 is the default. Because Windows printer output is usually sent directly to the printer, EMF and BMP files are not viewable. On UNIX, the printer output is directed to a file named `xprinter.eps` by default. For more information on printing views, scenes, or viewgroups, see "IDLgrPrinter::Draw" (*IDL Reference Guide*).

Positioning Objects Within a Page

Objects can be positioned in a printed page by first determining the size of the page. Use the IDLgrPrinter object DIMENSIONS property to return the size of the "drawable" area of the page. You can then use these dimensions to draw a view of specified dimensions in the center of the printed page. The following two examples show positioning objects within the printed page:

- The first example scales an orb object based on the page size and draws the view containing the orb to the center of the hardcopy page. See "Example: Centering an Orb" on page 280.
- The second example creates two IDLgrAxis objects and an orb object, each with a UNITS property value set to centimeters. The view is positioned in the center of the page, but the other object locations are specified in centimeters and drawn to the view in precise positions. See "Example: Precisely Positioning Vector and Bitmap Output" on page 281.

Example: Centering an Orb

The following example positions a view containing an orb object in the center of a page when it is printed. Centering the view is a common task. Using this example as a guideline, you can easily adapt it to meet your own needs.

```
PRO center_doc

; Define dimensions in centimeters (cm).
dims = [5.0, 5.0]

; Create a view with centimeters as units. Add the view to a model.
oView = OBJ_NEW('IDLgrView', $
   UNITS=2, $
   VIEWPLANE_RECT=[-dims[0]/2, -dims[1]/2, dims[0], dims[1]], $
   ZCLIP=[MAX(dims), -MAX(dims)], EYE=MAX(dims)+1, $
   COLOR=[200,200,200])
oModel = OBJ_NEW('IDLgrModel')
oView->Add, oModel

; Create an orb object and add it to the model.
oOrb1 = OBJ_NEW('orb', COLOR=[0,255,0], SHADING=1, $
   STYLE=2, HIDDEN=0)
oModel->Add, oOrb1

; Make radius 40% of window width.
oModel->Scale, dims[0]*0.4, dims[0]*0.4, dims[0]*0.4
oModel->Rotate, [1,1,0], 10

; Create a light and add it to the model.
oLight = OBJ_NEW('IDLgrLight', TYPE=1, LOCATION=[1.5,1.5,2])
oModel->Add, oLight

; Create a printer object, setting centimeters as the units.
oPrinter=OBJ_NEW('IDLgrPrinter', UNITS=2)

; Retrieve the drawable area of the page in the pagesize
; variable and use this to position the view.
oPrinter->GetProperty, DIMENSIONS=pageSize
centering = ((pageSize - dims)/2.)
oView->SetProperty, LOCATION=centering, DIMENSIONS=dims

; Print the view.
oPrinter->Draw, oView, VECTOR=1

OBJ_DESTROY,[oPrinter]
OBJ_DESTROY,[oView]

END
```

The following figure shows a subset of the output. The orb is positioned in the center of a printed page when you run this example.

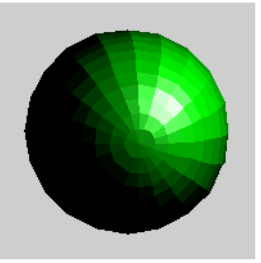


Figure 12-1: Output Centered in Printed Page

Example: Precisely Positioning Vector and Bitmap Output

The following example creates a model and draws some IDLgrAxis objects to the printer in vector mode. It then creates a second model for an orb object and plots the orb, drawing it to the printer in bitmap mode. The entire view is centered in the page, as shown in the previous example. However, this example precisely positions the orb and axes within the view using data units (defined as centimeters).

```
PRO center2_doc

; Set the view dimensions in units of centimeters (cm).
viewDims = [10.0, 10.0]

; Set the orb origin in cm, relative to the lower left
; corner of the view.
orbLoc = [3.0, 4.0]

; Set the Orb radius in cm.
orbRadius = 2.2

; Create the Orb object.
; The Orb object creates a unit orb with a default radius of 1.
oOrbModel = OBJ_NEW('IDLgrModel')
oOrb = OBJ_NEW('orb', COLOR=[0,255,0], SHADING=1, STYLE=2)
oOrbModel->Add, oOrb

; Create axes model. Create and position the axis objects.
oAxesModel = OBJ_NEW('IDLgrModel')
oX = OBJ_NEW('IDLgrAxis', 0, RANGE=[1,viewDims[0]-1], $
   /EXACT, LOCATION=[orbLoc[0]-orbRadius, 1])
```

```
oAxesModel->Add, oX
oY = OBJ_NEW('IDLgrAxis', 1, RANGE=[1, viewDims[1]-1], $
   /EXACT, LOCATION=[1, orbLoc[1]-orbRadius])
oAxesModel->Add, oY

; Add a box to show view extent.
oAxesModel->Add, OBJ_NEW('IDLgrPolygon', $
   [0, viewDims[0], viewDims[0], 0], $
   [0, 0, viewDims[1], viewDims[1]], STYLE=1)

; Create the view using the previously defined dimensions.
oView = OBJ_NEW('IDLgrView', $
   UNITS=2, VIEWPLANE_RECT=[0, 0, viewDims[0], viewDims[1]], $
   ZCLIP=[MAX(viewDims), -MAX(viewDims)], EYE=MAX(viewDims)+1, $
   COLOR=[255,255,255])
oTopModel = OBJ_NEW('IDLgrModel')
oView->Add, oTopModel

; Add a light.
oLight = OBJ_NEW('IDLgrLight', TYPE=1, LOCATION=[1.5,1.5,2])
oTopModel->Add, oLight

; Set up printer to print user-requested view. Center
; entire printer output in the page.
oPrinter=OBJ_NEW('IDLgrPrinter', UNITS=2)
oPrinter->GetProperty, DIMENSIONS=pageSize
centering = ((pageSize - viewDims)/2.)
oView->SetProperty, LOCATION=centering, DIMENSIONS=viewDims

; Print view containing axes in vector mode then remove model.
oTopModel->Add, oAxesModel
oPrinter->Draw, oView, VECTOR=1
oTopModel->Remove, oAxesModel

; Now float the orb into the view and print it in bitmap mode.
oTopModel->Add, oOrbModel
oView->SetProperty, VIEWPLANE_RECT = $
   [-orbRadius, -orbRadius, 2 * orbRadius, 2 * orbRadius], $
   LOCATION=[orbLoc[0]-orbRadius,orbLoc[1]-orbRadius]+centering, $
   DIMENSIONS=[2*orbRadius, 2*orbRadius]
   oPrinter->Draw, oView, VECTOR=0

; oPrinter->NewDocument
OBJ_DESTROY,[oPrinter]
OBJ_DESTROY,[oView]

END
```

The following figure shows a subset of the output. The entire plot area is positioned in the center of a printed page when you run this example.

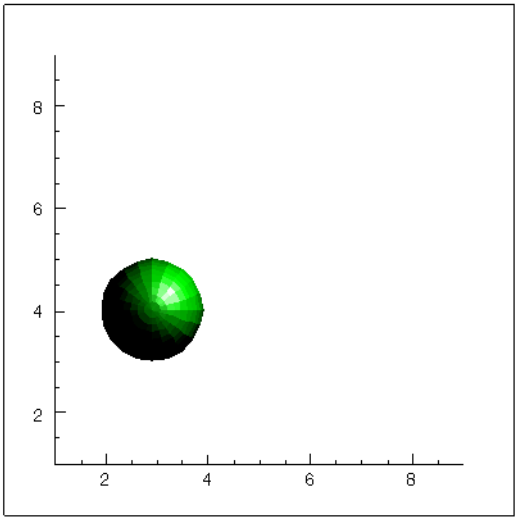


Figure 12-2: Positioning Objects Within a Printed Page

Starting a New Page on a Printer

To ensure that any subsequent calls to the IDLgrPrinter::Draw method occur on a new page, call the IDLgrPrinter::NewPage method:

```
myPrinter->NewPage
```

Submitting a Printer Job

To submit a printer job, call the IDLgrPrinter::NewDocument method. This method submits the printing job (consisting of all previous calls to IDgrPrinter::Draw and IDLgrPrinter::NewPage) to the printer.

After this method has been called, the printer is prepared to accept a new batch of graphics calls (via IDLgrPrinter::Draw).

```
myPrinter->NewDocument
```

Bitmap and Vector Graphic Output

The IDLgrClipboard and IDLgrPrinter destination objects allow objects in a scene, viewgroup, or view to be output as vector or bitmap graphics. Which output is suitable depends upon the contents of the scene being sent to the output destination object. Understanding the difference between bitmap and vector graphics will help clarify why there is a difference in how the final output is displayed, and how the output can be edited.

Bitmap Graphics

Bitmaps are a collection of bits that describe the individual pixels within an image. Each pixel is a specific color, and the matrix of these pixels compose the image. In bitmap graphics, the contents of a view, viewgroup. or scene are captured as an image and are drawn with pixels in the bitmap. They can be edited only by altering individual pixels. The following figure shows the individual pixels that are visible when a small segment of an image is greatly enlarged.

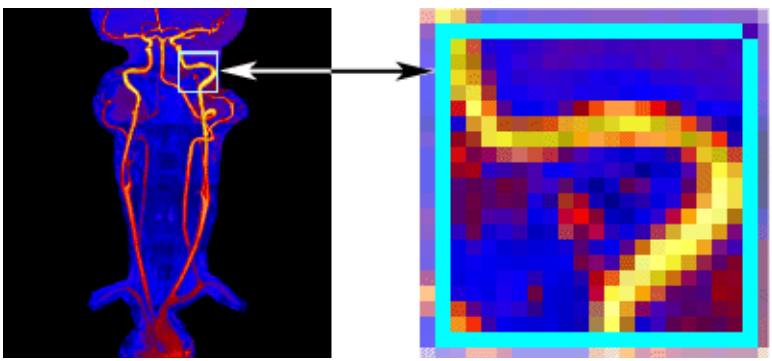


Figure 12-3: Sample Bitmap Image

IDLgrClipboard bitmap graphic output can be edited by any pixel-based paint program. In IDL, bitmap graphics can be stored as Bitmap (BMP）or PostScript (EPS) files under Windows, and as PostScript files under UNIX. Characteristically, bitmaps are large files, and image quality degrades when the image is substantially enlarged or reduced.

Vector Graphics

Vector graphics are described by simple graphic primitives. In the following figure, the vector output of the plot, shown on the left, is composed of multiple individual line segments that are defined mathematically. The IDLgrText objects are rendered as text primitives. All these primitives can be edited in vector graphic files. For example, in the following figure the final line segments in the plot have been repositioned in the right-hand image.

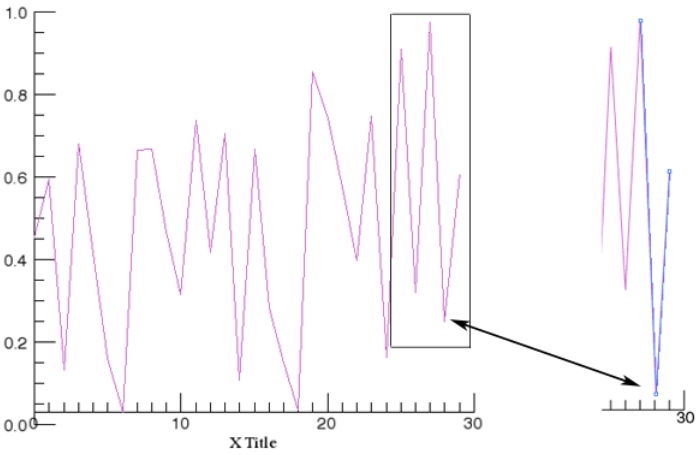


Figure 12-4: Sample Vector Image

IDLgrClipboard vector graphic output can be edited by an object-based graphical editor. In IDL, vector graphics can be stored as Enhanced MetaFile (EMF）or Encapsulated PostScript (EPS) files under Windows, and as Encapsulated PostScript (EPS) files under UNIX.

The main advantages of vector graphics are excellent scalability, and the ability to easily edit text and graphic features of the objects in the display. The graphic quality is maintained regardless of whether the graphic size is increased or decreased. The capabilities of the graphic editor determines what can be successfully edited. Simple lines and horizontal text can be easily edited in an EMF file inserted into a Microsoft Word document. However, more sophisticated graphic editors provide support for editing intricate graphic features and non-horizontal text. See “Text Rendering in Vector Graphics” on page 287 for more information. Vector graphics file sizes are generally smaller compared to bitmap graphics.

Guidelines for Choosing Bitmap or Vector Graphics

Advanced 3-D graphics rendering system output does not always map perfectly to a 2-D vector graphics system. The vector output is an approximation of what is displayed on the screen. How closely the vector output matches what is displayed depends upon the scene contents. Vector output may differ dramatically from bitmap output, and may also differ between the vector file formats (Encapsulated PostScript, Xprinter, and Enhanced MetaFile).

In general, scenes containing multiple, intersecting surfaces with various shading, transparency and lighting definitions are displayed with greater accuracy in a bitmap format than a vector format. However, simple 2-D plots are perfectly suited to vector output. Views containing the following items should not be output to vector graphic files:

- Transparent or semi-transparent objects — transparent objects in a view are not rendered in vector graphic files. Semi-transparent objects are rendered fully opaque.
- Textured or patterned objects — surfaces and polygons with textures or patterns are rendered without their textures or patterns.
- Hidden lines – polygon and surface objects drawn with the HIDDEN_LINES property set may experience missing lines.
- Volumes — volumes, other than those drawn in low quality wire frame mode (where the destination device QUALITY=0), are not rendered.
- Clipped objects — text strings and image objects do not appear clipped by clipping planes in vector graphic files. These objects only appear clipped by view boundaries.
- Smoothly shaded polygons and surfaces — Gouraud (smooth) shaded IDLgrPolygon and IDLgrSurface objects are displayed with smooth shading only in vector PostScript files generated by IDLgrClipboard, not in Enhanced MetaFile (EMF) vector format files, or in IDLgrPrinter vector EPS files. Polygons and surfaces appear with flat shading in EMF files and when printed.
- Lines and text in Xprinter — line style dash length is limited, and line style patterns cannot start and end with a ‘1’ bit when vector output is generated by Xprinter under UNIX. Also, text is always drawn as a set of triangles in Xprinter vector output, and cannot be edited.
- Objects dependent on depth buffering — depth buffering controls are not respected in vector graphic files. See “Primitive Object Sorting in Vector Graphics” on page 289 for more information.

Controlling What is Displayed in Vector Graphics

Several factors beyond the differences between bitmap and vector graphics (described in the previous section) affect a vector graphics file in terms of content and the ability to edit text. Keywords provide control over factors such as object sorting, polygon shading, and text rendering when using the Draw method of the IDLgrClipboard or IDLgrPrinter destination objects. See the following sections for more information:

- "Smooth Shading in Vector Graphics" in the following section
- "Text Rendering in Vector Graphics" on page 287
- "Primitive Object Sorting in Vector Graphics" on page 289

Smooth Shading in Vector Graphics

The IDLgrClipboard Draw method supports the VECT_SHADING keyword, which affects the appearance of the surfaces and polygons when the VECTOR and POSTSCRIPT keywords have also been set. When SHADING=1 (Gouraud shading) for IDLgrSurface or IDLgrPolygon, use this keyword to control the rendering quality. Set the VECT_SHADING keyword to one of the following:

- 0 = disable smooth shading. Setting this keyword causes all polygons and surfaces to be rendered with flat shading. This will override the SHADING value assigned to a surface or polygon object. This may be valuable when using slower PostScript interpreters.
- 1 = enable smooth shading. Setting this keyword renders smoothly shaded polygons in the Encapsulated PostScript file. This is the default.

Note

Polygons and surfaces in Enhanced MetaFiles (EMF) will be rendered using flat shading. Only the output in Encapsulated PostScript (EPS) files is affected by this keyword, and only when the VECTOR keyword has been set.

Text Rendering in Vector Graphics

Text can be easily edited in vector graphic files when the text is output as text primitives. In bitmap files, text glyphs cannot be edited except by modifying individual pixels. In a vector graphic file, IDLgrText objects are rendered as graphic primitives that can be edited. The IDLgrClipboard or IDLgrPrinter VECT_TEXT_RENDER_METHOD keyword controls whether text appears as filled

triangles or text primitives when the VECTOR keyword is also set. Set the VECT_TEXT_RENDER_METHOD keyword to one of the following:

- 0 = render text as text primitives. This uses the output device's text primitives when rendering text. This allows the text to be edited by object-based graphics programs. This is the default.
- 1 = render text as triangles. This produces text glyphs that closely match the text on the display device. The output file size is larger and contains filled triangles to represent text. This can preserve backward compatibility with the display of text objects prior to IDL 6.1, which introduced text primitives.

Note

When using the IDLgrPrinter object under UNIX, the Xprinter output is regarded as write-only. As there is no support for 3-D text, IDL always generates filled triangles when rendering text in the Xprinter output.

Setting VECT_TEXT_RENDER_METHOD=0 creates a vector graphics file with text rendered as primitives. The text associated with the graphic can be scaled, transformed or repositioned when edited in an object-oriented graphics application.

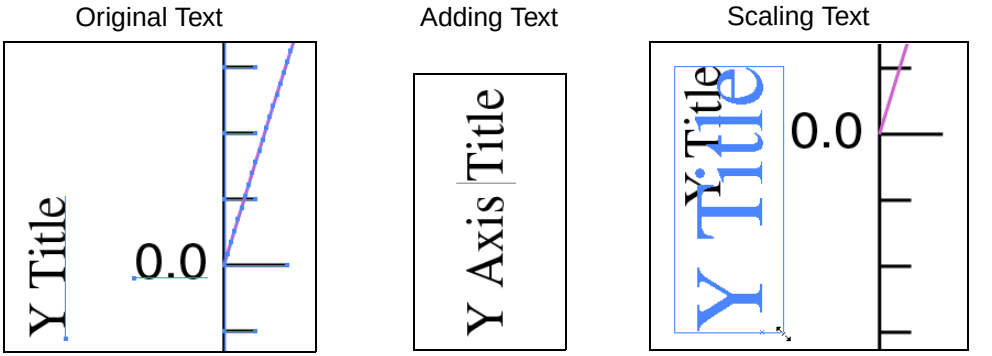


Figure 12-5: Editing Text Objects Output as Vector Graphics

An Enhanced MetaFile (EMF) inserted into a Microsoft Word document can be edited. However, not all versions of Microsoft Word support advanced 3-D graphic primitives such as those associated with obliquely or vertically aligned text. Choosing to edit a file with non-horizontally aligned text may result in the text being flattened into two dimensions. Typically, each letter becomes its own string and alignment is altered. To edit non-horizontal text and preserve the original quality,

create an Encapsulated PostScript (EPS) file that can be modified in a more sophisticated object-oriented image editing program.

Setting VECT_TEXT_RENDER_METHOD=1 creates text that is rendered as filled triangles. Elements of the plot in the following figure are composed of line segments that can be edited, but the text characters cannot be individually edited. The triangles composing the letters of the text object are visible in the right-hand image.

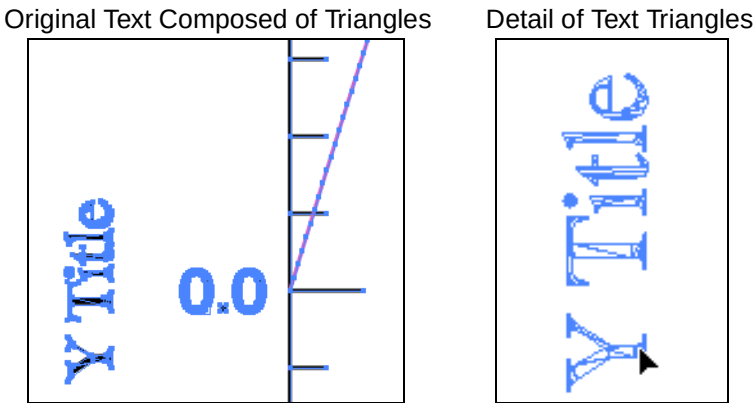


Figure 12-6: Text Objects Output as Triangles

Primitive Object Sorting in Vector Graphics

The IDLgrPrinter and IDLgrClipboard Draw methods support the VECT_SORTING keyword, which affects the appearance of the output when the VECTOR keyword has also been set. Use this keyword to simulate the depth buffer in Object Graphics in the output vector graphics file. Set the VECT_SORTING keyword to one of the following:

- 0 = disable sorting. The object primitives appear in the vector output file in the same order they are drawn on the display device. This is the order in which they appear in the graphics tree.
- 1 = enable sorting. Objects are ordered from back to front based on each primitive object's average depth value. This is the default.

The following figure shows the results of changing the VECT_SORTING keyword. When sorting is disabled (VECT_SORTING=0) as in the left image, the first object added to the model is drawn first in the display and in the destination device. In the code used to create the left image, the text is added to the model before the surface. Therefore it appears behind the surface in the vector graphics file. When the order is

reversed, the text is drawn on top of the surface. When sorting is enabled (VECT_SORTING=1) as in the right image, primitive objects are sorted according to their depth in the view. Most distant objects are drawn first. When two objects have the same average depth, the object added to the model first is drawn first and will appear behind subsequent objects.

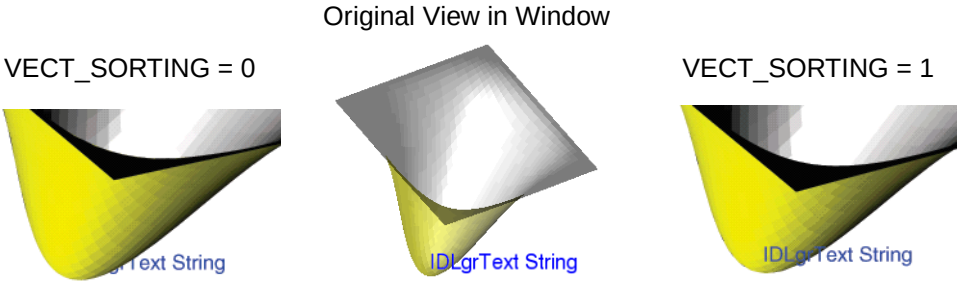


Figure 12-7: Controlling the Sorting of Object Primitives

Note

Vector output does not support depth test functions. Vector output resolves Z (depth) ties by using the DEPTH_TEST_FUNCTION default LESS depth test.

There are two instances in which the above sorting model is not applicable:

- In a window containing overlapping, transparent views
- In a window containing IDLgrImage objects

See the following sections for details.

Sorting Issues with Transparent Views

When a window contains multiple views, the objects in each view are sorted as a separate group. This simulates the default clear operation that IDL performs when drawing each view to a destination, clearing the depth buffer and repainting the view with the view color. Depending upon the ordering and transparency of the views, the vector output might not match what is displayed, regardless of the value of VECT_SORTING. Consider objects in a transparent view that are positioned behind an object in a non-transparent view. In the display, objects in the transparent view are occluded by the object that appears closer to the viewer. However, in the vector output, the objects in the transparent view interact with and are visible in the output. This occurs because IDL does not clear the depth buffer or repaint the view when it is transparent.

In the simple example shown in the following figure, the IDLgrText object is added to a transparent view and is positioned behind the surface. The view associated with the IDLgrSurface is not transparent. The view containing the surface and the transparent view containing the text are added to an IDLgrViewgroup and displayed in the window. The left image shows the vector file output, and the right image shows the bitmap file output. In the vector output, all of the text is visible because the views are sorted independently. This behavior occurs because the transparent view containing the text is added to the viewgroup after the view containing the surface. If the view containing the text is added first, then only the surface (whose view is not transparent) is drawn.

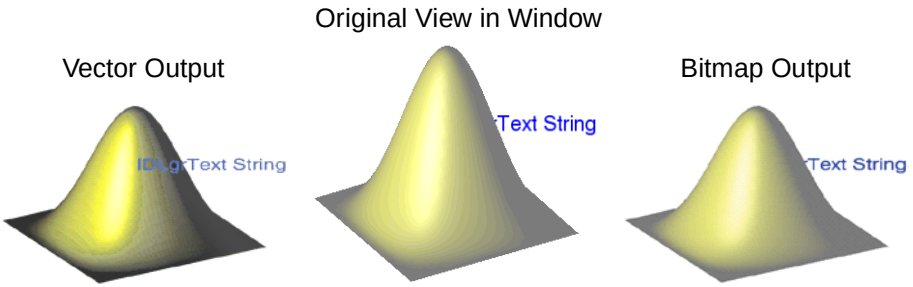


Figure 12-8: Interaction of Object Primitives with Transparent Views

Transparent Images

When IDL draws a semi-transparent image with vector output, it must draw it completely opaque, as it does with other primitives. Therefore, if you use image layers, where one image is semi-transparent in order to let you see another image drawn before it, the output will not be correct with vector output since the semi-transparent image will be drawn opaquely, completely hiding the image drawn before it. You should use bitmap output to get the desired results because semi-transparent rendering is not available with vector output.

Note

As described in "Guidelines for Choosing Bitmap or Vector Graphics" on page 286, all transparent objects (not just image objects) are rendered opaque in vector output.

Sorting Issues Among Image and Non-Image Objects

On a display device, IDLgrImage objects are drawn as "pixel primitives," which means that they do not update the depth buffer when they are written to the screen and also are not tested against the depth buffer to determine if they should be drawn or not by default. In such a case, images are rendered at Z=0 in viewing coordinates. This means:

- Images always overwrite any graphical data on the screen in the area in which they are drawn, regardless of their relative depth in the scene. Even objects that are rendered closer to the viewer than the image are overwritten.
- Objects that are drawn after an image is on the screen are drawn as if the image was not there. Since rendering the image did not update the depth buffer in the region where the image was rendered, the objects drawn after the image are not depth-tested against the image. This means that if you render an object, after rendering an image, so that it appears deeper than the image (Z < 0 in viewing coordinates), the object will render "on top" of the image, even though it is physically behind it in the scene.

Note

This is true unless you specifically enable depth testing (see "DEPTH_TEST_DISABLE" (*IDL Reference Guide*) for details). When depth testing is enabled, images behave just like any other 3-D object that supports depth buffer controls.

For these reasons, IDL applications often place image objects in the graphics tree so that they render first, unless the application wishes to make use of the behaviors described in the above two points. IDL emulates this behavior with vector graphics when VECT_SORTING is on as follows:

- Image objects are drawn in the order that they are positioned in the graphics tree.
- Non-image objects positioned before, after, or between image objects in the graphics tree are sorted amongst themselves. That is, non-image objects that are positioned in the tree before the first image are sorted and drawn first. Then the image is drawn. Then the next group of non-image objects are sorted and drawn, etc.

These steps assure consistency between bitmap and vector output for overlapping image and non-image primitives. However, some sorting differences may occur between non-image primitives that overlap each other but do not overlap images. For example, consider two non-image primitives drawn on the screen so that they do not

overlap an image, and one of these primitives is positioned in the graphics tree before (drawn before) the image, and the other is positioned in the graphics tree after (drawn after) the image. These two primitives are not sorted with respect to each other and are always drawn so that the second primitive is drawn after the first, regardless of their relative depth in the scene. If these primitives overlap, the result may not be correct if the first primitive is closer to the viewer than the second. Again, in this case, consider using bitmap output for more accurate output.

Chapter 13

Creating Custom Objects in IDL

The following topics are covered in this chapter:

Creating Custom Objects

This chapter describes the underlying structure of IDL objects and provides the information needed to create a custom object in IDL. This includes information on the object lifecycle, object methods (defining, using, and overriding methods) and custom object examples.

If you are creating objects in iTools, the concepts covered in this chapter are applicable, but you should use the *iTool Programming* as your reference when creating custom iTools, or iTool components. The *iTool Programming* provides information and examples of each of the major iTool elements (such as file readers and writers, manipulators, operations, and visualizations), and contains valuable discussions on data and property management within the iTool system.

IDL Object Overview

IDL objects are actually special *heap variables*, which means that they are global in scope and provide explicit user control over their lifetimes. Object heap variables can only be accessed via object references. Object references are discussed in this chapter. Heap variables in general are discussed in detail in "Heap Variables" (*Application Programming*).

Briefly, IDL provides support for the object concepts and mechanisms discussed in the following sections.

Classes and Instances

IDL objects are created as *instances* of a *class*, which is defined in the form of an IDL structure. The name of the structure is also the class name for the object. The *instance data* of an object is an IDL structure contained in the object heap variable, and can only be accessed by special functions and procedures, called *methods*, which are associated with the class. Class structures are discussed in "Creating an Object Class Structure" on page 300.

Encapsulation

Encapsulation is the ability to combine data and the routines that affect the data into a single object. IDL accomplishes this by only allowing access to an object's instance data via that object's *methods*. Data contained in an object is hidden from all but the object's own methods.

Methods

IDL allows you to define method procedures and functions using all of the programming tools available in IDL. Method routines are identified as belonging to an object class via a routine naming convention. Methods are discussed in detail in "Creating Custom Object Method Routines" on page 310.

Polymorphism

Polymorphism is the ability to create multiple object types that support the same operations. For example, many of IDL's graphics objects support an operation called "Draw," which sends graphics output to a specified place. The "Draw" operation is different in different contexts; sending a graphic to a printer is different from writing

it to a file. Polymorphism allows the details of the differences to remain hidden—all you need to know is that a given object supports the “Draw” operation.

Inheritance

Inheritance is the ability of an object class to inherit the behavior of other object classes. This means that when writing a new object class that is very much like an existing object class, you need only program the functions that are different from those in the inherited class. IDL supports multiple inheritance—that is, an object can inherit qualities from any number of other existing object classes. Inheritance is discussed in detail in “Inheritance” on page 302.

Persistence

Persistence is the ability of objects to remain in existence in memory after they have been created, allowing you to alter their behavior or appearance after their creation. IDL objects persist until you explicitly destroy them, or until the end of the IDL session. In practice, object persistence removes the need (in traditional IDL programs) to re-execute IDL commands that create an item (a plot, for example) in order to change a detail of the item. For example, once you have created a graphic object containing a plot, you can alter any aspect of the plot “on the fly,” without re-creating it. Similarly, having created an object containing a plot, you need not recreate the plot in order to print, save to an image file, or re-display it.

IDL objects also persist in the sense that you can use the SAVE and RESTORE routines to save and recreate objects between IDL sessions.

Undocumented Object Classes

Several of IDL's graphics objects are subclassed from more generic IDL objects. You may see references to the generic IDL objects when using IDL's HELP procedure to get information on an object, or when you use the OBJ_ISA or OBJ_CLASS functions. You may also notice that the generic objects are not documented in the "Object Class and Method Reference" (*IDL Reference Guide*). This is not an oversight.

We have chosen not to document the workings of the more generic objects from which the IDL graphics objects are subclassed because we reserve the right to make changes to their operation. We strongly recommend that you do not use the undocumented object classes directly, or subclass your own object classes from them. ITT Visual Information Solutions does not guarantee that user-written code that uses undocumented features will continue to function in future releases of IDL.

Creating an Object Class Structure

Object instance data is contained in named IDL structures. We will use the term *class structure* to refer to IDL structures containing object instance data.

Beyond the restriction that class structures must be named structures, there are no limits on what a class structure contains. Class structures can include data of any type or organization, including pointers and object references. When an object is created, the name of the class structure becomes the name of the class itself, and thus serves to define the names of all methods associated with the class. For example, if we create the following class structure:

```
struct = { Class1, data1:0L, data2:FLTARR(10) }
```

any objects created from the class structure Class1 would have the same two fields (data1, a long integer, and data2, a ten-element floating-point array) and any methods associated with the class would have the name Class1::*method*, where *method* is the actual name of the method routine. Methods are discussed in detail in “Creating Custom Object Method Routines” on page 310.

Note

When a new instance of a structure is created from an existing named structure, all of the fields in the newly-created structure are *zeroed*. This means that fields containing numeric values will contain zeros, fields containing string values will contain null strings, and fields containing pointers or objects will contain null pointers or null objects. In other words, no matter what data the original structure contained, the new structure will contain only a template for that type of data. This is true of objects as well; a newly created object will contain a zeroed copy of the class structure as its instance data.

It is important to realize that creating a class structure does not create an object. Objects can only be created by calling the OBJ_NEW or OBJARR function with the name of the class structure as the argument, and can only be accessed via the returned object reference. In addition, object methods can only be called on objects, and not on class structures themselves.

Once defined, a given class structure type cannot be changed. If a structure definition is executed and the structure already exists, each tag name and the structure of each tag field must agree with the original definition. To redefine a structure, you must either reset or exit the current IDL session.

Automatic Class Structure Definition

If IDL finds a reference to a structure that has not been defined, it will search for a structure definition procedure to define it. (This is true of all structure references, not just class structures.) Automatic structure definition is discussed in “Automatic Structure Definition” on page 352. Briefly, if IDL encounters a structure reference for a structure type that has not been defined, it searches for a routine with a name of the form

```
STRUCT__DEFINE
```

where STRUCT is the name of the structure type. Note that there are *two* underscores in the name of the structure definition routine.

The following is an example of a structure definition procedure that defines a structure that will be used for the class CNAME.

```
PRO CNAME__DEFINE
    struct = { CNAME, data1:0L, data2:FLTARR(10) }
END
```

This defines a structure named CNAME with 2 data fields (data1, a long integer, and data2, a ten-element floating-point array). If you tell IDL to create an object of type CNAME before this structure has been defined, IDL will search for the procedure CNAME__DEFINE to define the class structure before attempting to create the object. If the CNAME__DEFINE procedure has not yet been compiled, IDL will use its normal routine searching algorithm to attempt to find a file named CNAME__DEFINE.PRO. If IDL cannot find a defined structure or structure definition routine, the object-creation operation will fail.

Note

If you are creating structure definitions on the fly, the possibility exists that you will run into namespace conflicts — that is, a structure with the same name as the structure you are attempting to create may already exist. This can be a problem if you are developing object-oriented applications for others, since you probably do not have much control over the IDL environment on your clients’ systems. You can avoid most problems by creating a unique namespace for your routines; ITT Visual Information Solutions does this by prefixing the names of objects with the letters “IDL”. To help avoid namespace conflict, consider using a custom prefix (not “IDL”). To be completely sure that the objects created by your programs are what you expect, however, you should have the program inspect the created structures and handle errors appropriately.

Inheritance

When defining a class structure, use the INHERITS specifier to indicate that this structure inherits instance data and methods from another class structure. For example, if we defined a class structure called “circle,” as follows:

```
struct = { circle, x:0, y:0, radius:0 }
```

we can define a subclass of the “circle” class like this:

```
struct = { filled_circle, color:0, INHERITS circle }
```

You can use the INHERITS specifier in any structure definition. However, when the structure being defined is a *class structure* (that is, an object will be created from the structure), inheritance affects both the structure definition and the object methods available to the object that inherits. The INHERITS specifier is discussed in “Structure Inheritance” on page 338.

When a class structure inherits from another class structure, it is said to be a *subclass* of the class it inherits from. Similarly, the class that is inherited from is called a *superclass* of the new class. Defining a subclass of an existing class in this manner has two consequences. First, the class structure for the subclass is constructed as if the elements of the inherited class structure were included in-line in the structure definition. In our example, the command defining the “filled_circle” class above would create the followings structure definition:

```
{ filled_circle, color:0, x:0, y:0, radius:0 }
```

Note that the data fields from the inherited structure definition appear in-line at the point where the INHERITS specifier appears.

The second consequence of defining a subclass structure that inherits from another class structure is that when an object is created from the subclass structure, that object inherits the *methods* of the superclass as well as its data fields. That is, if an object of the superclass type has a method, that method is available to objects created from the subclass as well. In our example above, say we create an object of type circle and define a Print method for it. Any objects of type filled_circle will also have access to the Print method defined for circle.

IDL allows multiple inheritance. This means that you can include the INHERITS specifier as many times as you desire in a structure definition, *as long as all of the resulting data fields have unique names*. Data fields must have unique names because when the class structure definition is built, the tag names are included in-line at the point where the INHERITS specifier appears. Duplicate tag names will cause the structure definition to fail; it is your responsibility as a programmer to ensure that tag names are not used more than once in a structure definition.

Note

The requirement that names be unique applies only to *data* fields. It is perfectly legitimate (and often necessary) for subclasses to have methods with the same names as methods belonging to the superclass. See "Method Overriding" on page 314 for details.

If a structure referred to by an INHERITS specifier has not been defined in the current IDL session, IDL will attempt to define it in the manner described in "Automatic Class Structure Definition" on page 301.

Null Objects

The *Null Object* is a special object reference that is guaranteed to never point at a valid object heap variable. It is used by IDL to initialize object reference variables when no other initializing value is present. It is also a convenient value to use when defining structure definitions for fields that are object references, since it avoids the need to have a pre-existing valid object reference.

Null objects are created when you call an object-creation routine but do not specify a class structure to be used as the new object's template. The following statement creates a null object:

```
nullobj = OBJ_NEW()
```

Object Heap Variables

Object heap variables are IDL heap variables that are accessible only via *object references.* While there are many similarities between object references and pointers, it is important to understand that they are not the same type, and cannot be used interchangeably. Object heap variables are created using the OBJ_NEW and OBJARR functions. For more information on heap variables and pointers, see "IDL Pointers" on page 364.

Heap variables are a special class of IDL variables that have global scope and explicit user control over their lifetime. They can be basic IDL variables, accessible via pointers, or objects, accessible via object references. In IDL documentation of pointers and objects, heap variables accessible via pointers are called *pointer heap variables*, and heap variables accessible via object references are called *object heap variables*.

Note

Pointers and object references have many similarities, the strongest of which is that both point at heap variables. It is important to understand that they are not the same type, and cannot be used interchangeably. Pointers and object references are used to solve different sorts of problems. Pointers are useful for building dynamic data structures, and for passing large data around using a lightweight token (the pointer itself) instead of copying data. Objects are used to apply object oriented design techniques and organization to a system. It is, of course, often useful to use both in a given program.

Heap variables are global in scope, but do not suffer from the limitations of COMMON blocks. That is, heap variables are available to all program units at all times. (Remember, however, that IDL variables containing pointers to heap variables are *not* global in scope and must be declared in a COMMON block if you want to share them between program units.)

Heap variables:

- Facilitate object oriented programming.
- Provide full support for Save and Restore. Saving a pointer or object reference automatically causes the associated heap variable to be saved as well. This means that if the heap variable contains a pointer or object reference, the heap variables they point to are also saved. Complicated self-referential data structures can be saved and restored easily.

- Are manipulated primarily via pointers or object references using built in language operators rather than special functions and procedures.
- Can be used to construct arbitrary, fully general data structures in conjunction with pointers.

Dangling References

If a heap variable is destroyed, any remaining pointer variable or object reference that still refers to it is said to contain a *dangling reference*. Unlike lower level languages such as C, dereferencing a dangling reference will not crash or corrupt your IDL session. It will, however, fail with an error message.

There are several possible approaches to avoiding such errors. The best option is to structure your code such that dangling references do not occur. You can, however, verify the validity of pointers or object references before using them (via the PTR_VALID or OBJ_VALID functions) or use the CATCH mechanism to recover from the effect of such a dereferencing.

Heap Variable “Leakage”

Heap variables are not reference counted—that is, IDL does not keep track of how many references to a heap variable exist, or stop the last such reference from being destroyed—so it is possible to lose access to them and the memory they are using. See “Heap Variables” on page 359 for additional details.

Freeing Heap Variables

The HEAP_FREE procedure recursively frees all heap variables (pointers or objects) referenced by its input argument. This routine examines the input variable, including all array elements and structure fields. When a valid pointer or object reference is encountered, that heap variable is marked for removal, and then is recursively examined for additional heap variables to be freed. In this way, all heap variables that are referenced directly or indirectly by the input argument are located. Once all such heap variables are identified, HEAP_FREE releases them in a final pass. Pointers are released as if the PTR_FREE procedure was called. Objects are released as with a call to OBJ_DESTROY.

HEAP_FREE is recommended when:

- The data structures involved are highly complex, nested, or variable, and writing cleanup code is difficult and error prone.
- The data structures are opaque, and the code cleaning up does not have knowledge of the structure.

See "HEAP_FREE" (*IDL Reference Guide*) for further details.

The Object Lifecycle

Objects are *persistent*, meaning they exist in memory until you destroy them. We can break the life of an object into three phases: creation and initialization, use, and destruction. Object *lifecycle routines* allow the creation and destruction of object references; *lifecycle methods* associated with an object allow you to control what happens when an object is created or destroyed.

This section will discuss the first and last phases of the object lifecycle; the remainder of this chapter discusses manipulation of existing objects and use of object method routines. To get information about an object, see "Returning Object Type and Validity" (Chapter 4, *Using IDL*).

Creation and Initialization

Object references are created using one of two lifecycle routines: OBJ_NEW or OBJARR. Newly created objects are initialized upon creation in two ways:

1. The object reference is created based on the class structure specified,
2. The object's Init method (if it has one) is called to initialize the object's instance data (contained in fields defined by the class structure). If the object does not have an Init method, the object's superclasses (if any) are searched for an Init method.

The Init Method

An object's lifecycle method Init is a function named *Class*::Init (where *Class* is the actual name of the class). The purpose of the Init method is to populate a newly-created object with instance data. Init should return a scalar TRUE value (such as 1) if the initialization is successful, and FALSE (such as 0) if the initialization fails.

The Init method is unusual in that it *cannot be called outside an object-creation operation*. This means that—unlike most object methods—you cannot call the Init method on an object directly. You can, however, call an object's Init method from within the Init method of a subclass of that object. This allows you to specify parameters used by the superclass' Init method along with those used by the Init method of the object being created. In practice, this is often done using the _EXTRA keyword. See"Keyword Inheritance" on page 89 for details.

The OBJ_NEW Function

Use the OBJ_NEW function to create an object reference to a new object heap variable. If you supply the name of a class structure as its argument, OBJ_NEW creates a new object containing an instance of that class structure. Note that the fields of the newly-created object's instance data structure will all be empty. For example, the command:

```
obj1 = OBJ_NEW('ClassName')
```

creates a new object heap variable that contains an instance of the class structure *ClassName*, and places an object reference to this heap variable in obj1. If you do not supply an argument, the newly-created object will be a null object.

When creating an object from a class structure, OBJ_NEW goes through the following steps:

1. If the class structure has not been defined, IDL will attempt to find and call a procedure to define it automatically. See "Automatic Class Structure Definition" on page 301 for details. If the structure is still not defined, OBJ_NEW fails and issues an error.
2. If the class structure has been defined, OBJ_NEW creates an object heap variable containing a zeroed instance of the class structure.
3. Once the new object heap variable has been created, OBJ_NEW looks for a *method* function named *Class*::Init (where *Class* is the actual name of the class). If an Init method exists, it is called with the new object as its implicit SELF argument, as well as any arguments and keywords specified in the call to OBJ_NEW. If the class has no Init method, the usual method-searching rules are applied to find one from a superclass. For more information on methods and method-searching rules, see "Creating Custom Object Method Routines" on page 310.

Note

OBJ_NEW does not call all the Init methods in an object's class hierarchy. Instead, it simply calls the first one it finds. Therefore, the Init method for a class should call the Init methods of its direct superclasses as necessary.

4. If the Init method returns true, or if no Init method exists, OBJ_NEW returns an object reference to the heap variable. If Init returns false, OBJ_NEW destroys the new object and returns the NULL object reference, indicating that the operation failed. Note that in this case the Cleanup method is not called.

See "OBJ_NEW" (*IDL Reference Guide*) for further details.

The OBJARR Function

Use the OBJARR function to create an array of objects of up to eight dimensions. Every element of the array created by OBJARR is set to the null object. For example, the following command creates a 3 by 3 element object reference array with each element contain the null object reference:

```
obj2 = OBJARR(3, 3)
```

See "OBJARR" (*IDL Reference Guide*) for further details.

Destruction

Use the OBJ_DESTROY procedure to destroy an object. If the object's class, or one of its superclasses, supplies a procedure method named Cleanup, that method is called, and all arguments and keywords passed by the user are passed to it. The Cleanup method should perform any required cleanup on the object and return. Whether a Cleanup method actually exists or not, IDL will destroy the heap variable representing the object and return.

The Cleanup method is unusual in that it *cannot be called outside an object-destruction operation*. This means that—unlike most object methods—you cannot call the Cleanup method on an object directly. You can, however, call an object's Cleanup method from within the Cleanup method of a subclass of that object.

Note that the object references themselves are not destroyed. Object references that refer to nonexistent object heap variables are known as dangling references, and are discussed in more detail in "Dangling References" on page 371.

See "OBJ_DESTROY" (*IDL Reference Guide*) for further details.

Implicit Calling of Superclass Cleanup Methods

If you create an object class and do not implement a Cleanup method for it, when you destroy an object of your class IDL will call the Cleanup method of the class' superclass, if it has one.

If your class has multiple superclasses, on destruction IDL will attempt to call the Cleanup method of the first superclass. If that superclass has a Cleanup method, IDL will execute it and then destroy the object. If the first superclass does not have a Cleanup method, IDL will proceed through the list of superclasses in the order they are specified in the class structure definition statement until it either finds a Cleanup method to execute or reaches the end of the list.

To ensure that Cleanup methods from multiple superclasses are called, create a Cleanup method for your class and call the superclass' Cleanup methods explicitly.

Creating Custom Object Method Routines

IDL objects can have associated procedures and functions called *methods*. Methods are called on objects via their object references using the method invocation operator.

While object methods are constructed in the same way as any other IDL procedure or function, they are different from other routines in the following ways:

- Object methods are defined using a special naming convention that incorporates the name of the class to which the method belongs. See "Defining Method Routines" below.
- All method routines automatically pass an implicit argument named self, which contains the object reference of the object on which the method is called. See "The Implicit Self Argument" on page 311.
- Object methods cannot be called on their own. You must use the method invocation operator and supply a valid object reference, either of the class the method belongs to or of one of that class' subclasses. See "Calling Method Routines" on page 312.

Note

Keyword inheritance is an extremely important concept to understand when working with object methods. See "Keyword Inheritance" on page 89 for details.

Defining Method Routines

Method routines are defined in the same way as other IDL procedures and functions, with the exception that the name of the class to which they belong, along with two colons, is prepended to the method name:

```
PRO ClassName::Method
    IDL statements
END
```

or

```
FUNCTION ClassName::Method, Argument1
    IDL statements
RETURN, value
END
```

For example, suppose we create two objects, each with its own "print" method.

First, define two class structures:

```
struct = { class1, data1:0.0 }
struct = { class2, data2a:0, data2b:0L, INHERITS class1 }
```

Now we define two “print” methods to print the contents of any objects of either of these two classes. (If you are typing this at the IDL command line, enter the .RUN command before each of the following procedure definitions.)

```
PRO class1::Print1
    PRINT, self.data1
END
PRO class2::Print2
    PRINT, self.data1
    PRINT, self.data2a, self.data2b
END
```

Once these procedures are defined, any objects of class1 have access to the method Print1, and any objects of class2 have access to both Print1 and Print2 (because class2 is a subclass of—it *inherits* from—class1). Note that the Print2 method prints the data1 field inherited from class1.

Note

It is not necessary to give different method names to methods from different classes, as we have done here with Print1 and Print2. In fact, in most cases both methods would have simply been called Print, with each object class knowing only about its own version of the method. We have given the two procedures different names here for instructional reasons; see “Method Overriding” on page 314 for a more complete discussion of method naming.

The Implicit *Self* Argument

Every method routine has an implicit argument parameter named self. The self parameter always contains the object reference of the object on which the method is called. In the method routines created above, self is used to specify which object the data fields should be printed from using the structure dot operator:

```
PRINT, self.data1
```

You do not need to explicitly pass the self argument; in fact, if you try to specify an argument called self when defining a method routine, IDL will issue an error.

Calling Method Routines

You must use the method invocation operator (->) to call a method on an object. The syntax is:

ObjRef->Method

where *ObjRef* is an object reference and *Method* is a method belonging either to the object's class or to one of its superclasses. *Method* may be specified either partially (using only the method name) or completely using both the class name and method name, connected with two colons:

ObjRef->Class::Method

See "Specifying Class Names in Method Calls" on page 315 for more information.

The exact method syntax is slightly different from other routine invocations:

```
; For a procedure method.
ObjRef->Method

; For a function method.
Result = ObjRef->Method()
```

Where *ObjRef* is an object reference belonging to the same class as the *Method*, or to one of that class' subclasses. We can illustrate this behavior using the Print1 and Print2 methods defined above.

First, define two new objects:

```
A = OBJ_NEW('class1')
B = OBJ_NEW('class2')
```

We can call Print1 on the object A as follows:

```
A->Print1
```

IDL prints:

```
0.00000
```

Similarly, we can call Print2 on the object B:

```
B->Print2
```

IDL prints:

```
0.00000
0           0
```

Since the object B inherits its properties from class1, we can also call Print1 on the object B:

```
B->Print1
```

IDL prints:

```
0.00000
```

We cannot, however, call Print2 on the object A, since class1 does not inherit the properties of class2:

```
A->Print2
```

IDL prints:

```
% Attempt to call undefined method: 'CLASS1::PRINT2'.
```

Searching for Method Routines

When a method is called on an object reference, IDL searches for it as with any procedure or function, and calls it if it can be found, following the naming convention established for structure definition routines. (See "Automatic Class Structure Definition" on page 301.) In other words, IDL discovers methods as it needs them in the same way as regular procedures and functions, with the exception that it searches for files named

classname__method.pro

rather than simply

method.pro

Remember that there are two *underscores* in the file name, and two *colons* in the method routine's name.

Note

If you are working in an environment where the length of filenames is limited, you may want to consider defining all object methods in the same `.pro` file you use to define the class structure. This practice avoids any problems caused by the need to prepend the *classname* and the two underscore characters to the method name. If you must use different `.pro` files, make sure that all class (and superclass) definition filenames are unique in the first eight characters.

Method Overriding

Unlike data fields, method names can be duplicated. This is an important feature that allows method overriding, which in turn facilitates polymorphism in the design of object-oriented programs. Method overriding allows a subclass to provide its own implementation of a method already provided by one of its superclasses. When a method is called on an object, IDL searches for a method of that class with that name. If found, the method is called. If not, the methods of any inherited object classes are examined in the order their INHERITS specifiers appear in the structure definition, and the first method found with the correct name is called. If no method of the specified name is found, an error occurs.

The method search proceeds *depth first, left to right*. This means that if an object's class does not provide the method called directly, IDL searches through inherited classes by first searching the left-most included class—and all of its superclasses—before proceeding to the next inherited class to the right. If a method is defined by more than a single inherited structure definition, the first one found is used and no warning is generated. This means that class designers should pick non-generic names for their methods as well as their data fields. For example, suppose we have defined the following classes:

```
struct = { class1, data1}
struct = { class2, data2a:0, data2b:0.0, inherits class1 }
struct = { class3, data3:'', inherits class2, inherits class1 }
struct = { class4, data4:0L, inherits class2, inherits class3 }
```

Furthermore, suppose that both class1 and class3 have a method called Print defined.

Now suppose that we create an object of class4, and call the Print method:

```
A = OBJ_NEW('class4')
A->Print
```

IDL takes the following steps:

1. Searches class4 for a Print method. It does not find one.
2. Searches the left-most inherited class (class2) in the class definition structure for a Print method. It does not find one.
3. Searches any superclasses of class2 for a Print method. It finds the class1 Print method and calls it on A.

Notice that IDL stops searching when it finds a method with the proper name. Thus, IDL doesn't find the Print method that belongs to class3.

When are Methods Associated with Object Classes?

It is important to note that IDL will associate a method with objects of a given class the first time the method is called on an object of that class. This means that if a new method definition is compiled *after* the first time a method with a particular name is called, the new definition will not be used until a new IDL session begins.

Extending the example above, suppose that *after* calling the Print method you compile a new class4::Print method. Subsequent calls to the Print method will still invoke the class1::Print method even though the object instance A's "own" Print method now exists. Once an association has been formed between an object class and a method, that association is not changed for the duration of the IDL session.

To ensure that the correct method is selected, either ensure that the method is compiled before the first time it is called or explicitly specify the class name when calling the method, as described below.

Specifying Class Names in Method Calls

If you specify a class name when calling an object method, like so:

```
ObjRef->classname::method
```

Where *classname* is the name of one of the object's superclasses, IDL will search *classname* and any of *classname's* superclasses for the method name. IDL will *not* search the object's own class or any other classes the object inherits from.

This type of method call is especially useful when a class has a method that overrides a superclass method and does its job by calling the superclass method and then adding functionality. In our simple example from "Calling Method Routines" on page 312, above, we could have defined a Print method for each class, as follows:

```
PRO class1::Print
    PRINT, self.data1
END
PRO class2::Print
    self->class1::Print
    PRINT, self.data2a, self.data2b
END
```

In this case, to duplicate the behavior of the Print1 and Print2 methods, we make the following method calls:

```
A->Print
```

IDL prints:

```
0.00000
```

And now the B:

```
B->Print
```

IDL prints:

```
0.00000
0           0
```

Now we'll use the second method:

```
B->class1::Print
```

IDL prints:

```
0.00000
```

And now A:

```
A->class2::Print
```

IDL prints:

```
% CLASS2 is not a superclass of object class CLASS1.
% Execution halted at:  $MAIN$
```

Object Examples

We have included a number of examples of object-oriented programming as part of the IDL distribution. Many of the examples used in this volume are included — sometimes in expanded form — in the `examples/doc/objects` subdirectory of the IDL distribution. By default, this directory is part of IDL's path; if you have not changed your path, you will be able to run the examples as described here. See "!PATH" (*IDL Reference Guide*) for information on IDL's path.

Also see the *iTool Programming* for additional examples of creating custom objects including file reader and writers, manipulators, and operators that can be used within a custom iTool.

Creating Composite Classes or Subclasses

IDL includes a rich set of basic objects that an be used for creating visualizations. You may find that you are using a certain combination of these objects again and again within your applications for a particular purpose. If this is the case, you might want to consider defining a composite object class that encapsulates the combination of those subcomponents.

IDL includes several such composite classes, such as the IDLgrColorbar and IDLgrLegend objects. You will find the IDL code for these objects in the `lib` directory of your IDL distribution.

Example Code

Another example can be found in the `idlexshow3__define.pro` in the `examples/doc/utilities` subdirectory. In this case, an image, surface, and contour representation are combined into a single object called the IDLexShow3 object. To see this object being used in an application, run the show3_track routine, defined in the file `show3_track.pro` in the `examples/doc/objects` directory.

The program `show3_track.pro` creates the following visualization:

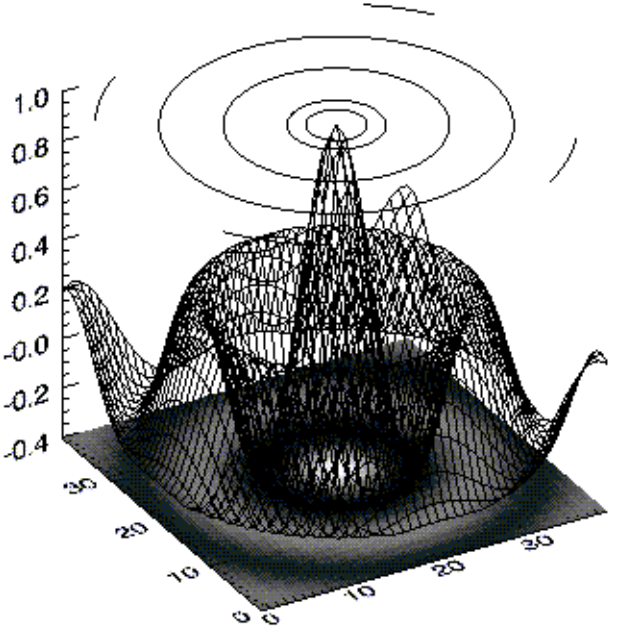


Figure 13-1: Show3_track example

You may also find that you want to customize one or more of the classes available in Object Graphics. For instance, you may want to create a specialized image object that can handle 16-bit palettes.

Example Code

An example that creates a specialized image object that can handle 16-bit palettes is provided in `idlexpalimage__define.pro` in the `examples/doc/utilities` subdirectory of the IDL distribution. Run the example procedure by entering `idlexpalimage__define` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT idlexpalimage__define.pro`.

Chapter 14

Advanced Rendering Using Shader Objects

The following chapter describes how to use IDLgrShader functionality to take advantage of graphics card processing and rendering capabilities in IDL object graphics applications.

About Shaders

The *shader* functionality implemented in IDL object graphics provides access to the advantages of the hardware-based OpenGL Shading Language (GLSL) features that are available on modern graphics cards. Using a shader, computationally intensive image processing operations can be off-loaded to the graphics card, making the time and processing resources of the host computer available to other application elements. Additionally, the OpenGL Shading Language greatly expands on the capabilities of the fixed OpenGL rendering pipeline to produce advanced visual effects. Whereas native IDL object graphics expose OpenGL capabilities through fixed object properties, GLSL offers the ability to modify virtually any object characteristic. Using shaders lets you implement realistic material and lighting effects, create animations by modifying object vertices, and achieve image processing performance rates that far exceed what is possible using the system CPU.

Note

It is important to realize that this functionality only exposes the ability to use OpenGL Shading Language within an IDL application. It does not implement the shading language nor does this document explain how to write shader language code. However, numerous GLSL publications and internet resources are available.

Why Use Shaders

Shaders are often used to produce elaborate scenes including realistic materials and lighting, especially in 3-D gaming environments. However, shaders also offer incredible performance and enhanced interactivity when used in image processing applications. Consider an application the applies the following operations to an image:

- Convolution filter
- Scale and offset bias
- Tonal compensation (LUT)
- Display compensation (LUT or BYTSCL)

Using the system CPU as the primary processor in a software-based solution, it is only possible to achieve a display rate of a few frames per second. However, if a shader program is implemented, the processing is shifted to the graphics card GPU and display rates of over 100 frames per second are possible. The shader program applies these operations on every draw so there is no performance penalty for altering parameters during rapid drawing sequence. This means that a user can change a

parameter of the operation and see the results nearly instantaneously, which makes the image processing application highly responsive to interactive changes. See "How Shaders Enhance Performance" on page 326 for details.

Shaders also provide a means of solving a wider range of image processing problems than what is possible using only the fixed functionality of the OpenGL pipeline exposed by IDL procedures and functions. There are no limits to the image processing problems that you can solve using shaders other than those imposed by the shader itself and the boundaries of your imagination.

Hardware Requirements for Shaders

In general, shader programs will work on graphics cards and drivers that support the OpenGL 2.0 interface. However, it is important to note that performance varies greatly between low-end and high-end graphics cards, and also varies depending on the implementation and content of the shader program. Also, always use the most up-to-date drivers available for your graphics card when developing IDL applications that use shader programs.

Use the SHADING_LANGUAGE_VERSION keyword to IDLgrWindow::GetDeviceInfo to determine whether or not a card supports shader functionality. Executing the following code in IDL will briefly create an IDLgrWindow object and report on whether hardware shaders are available on your system:

```
oWin = OBJ_NEW('IDLgrWindow')
oWin->GetDeviceInfo, SHADING_LANGUAGE_VERSION=v
OBJ_DESTROY, oWin
PRINT, 'Shading language version: ', v
IF FLOAT(v) GE 2 THEN PRINT, 'Hardware shaders are available' $
   ELSE PRINT, 'Hardware shaders are not available'
```

A shader-equipped graphics card will not utilize the shader hardware if IDL is using software rendering. To make sure you are using the shading hardware, be sure to specify the hardware renderer (for example, set the IDLgrWindow RENDERER property to 0).

Image processing applications can provide a software-based alternative in case the system graphics card does not support OpenGL 2.0. See "Providing a Software Alternative to Shaders" on page 335 for details.

Note

If there is insufficient support for the shader program, IDL draws the scene as if there was no shader object present unless a software fallback exists.

Note

Setting the IDLgrWindow property RETAIN to 2 disables hardware shaders. Software shaders are used if available.

About Shader Programs

A shader program is a user-defined program written in OpenGL Shading Language (GLSL) that is executed by the graphics processing unit (GPU) of the graphics card. This chapter provides an overview of the process of using shader programs with IDL. It is not meant as a tutorial on writing shader programs.

Note

Your graphics card must support OpenGL 2.0 functionality and you will need to have the latest drivers installed to take advantage of shader programs in IDL.

Shader programs can produce results that are not possible using the fixed-function rendering pipeline exposed through IDL object properties. For example, if you create an IDLgrPolygon and set the COLOR property to green and the SHADING property to flat, OpenGL takes over rendering of a green polygon with flat shading; more precise control is not possible. However, a shader program provides far more control and lets you configure lighting and texture effects on a per-pixel basis.

The interaction of a shader program within the graphics system is shown in the following figure. The graphics card GPU switches between executing fixed-function and shader program code.

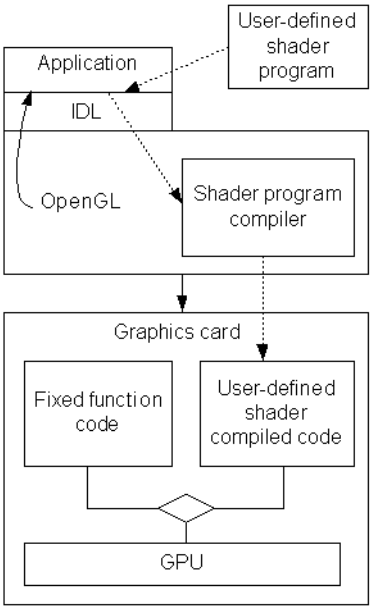


Figure 14-1: Shader Program Interaction with Application and Graphics Card

Note

Shader program attributes override all fixed-function attributes (those defined using object properties). If you define a blue sphere in IDL object graphics, but define a shader program to draw a green sphere, the displayed sphere will be green if there is suitable hardware support for the shader program.

Vertex and Fragment Shaders

Shader programs are highly configurable because each shader program consists of two required parts: a *vertex shader* and a *fragment shader*. (A fragment is the same thing as a pixel, but with extra information such as depth.) The shader program compiler built into OpenGL compiles each of them separately and then links them to form a complete shader program.

When a shader program is active, OpenGL calls the vertex shader program once for every vertex in the primitive it is currently drawing. Along with the expected position information (*x*, *y*, *z*, *w*) for the vertex, there is also color, normal, texture coordinate, lighting, and other information associated with the vertex that is available to the vertex shader program. Here, the vertices, connectivity and transformation information are used to construct the primitive. The primitive undergoes rasterization, which converts the vertex representation to pixel representation. This defines the fragments.

OpenGL calls the fragment shader for every pixel that OpenGL intends to modify on the graphics device. The fragment shader determines the color of the pixel according to information it may obtain from the vertex program and from its own calculations. The shader program may also include computing normals to apply per-fragment lighting effects. It then tells OpenGL what color to use for drawing the pixel.

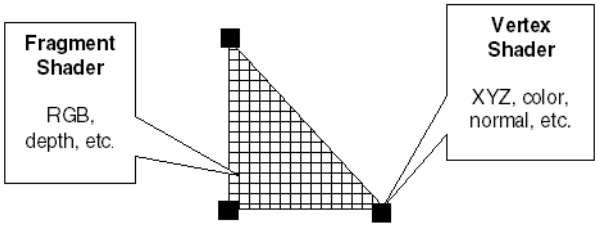


Figure 14-2: Interaction of Vertex and Fragment Shaders on a Primitive

For the primitive shown in the previous figure, OpenGL calls the vertex shader three times, once for each corner of the triangle, and calls the fragment shader program once for every pixel covered by the triangle. Vertex attributes are interpolated across the fragments based on the vertex connectivity and the resulting distance of a fragment from a vertex.

How Shaders Enhance Performance

Using a shader lets you take advantage of the processing power of the graphics card processing unit (GPU) instead of relying solely on the system CPU. The ability to offload computationally intensive tasks means applications run faster and operate more interactively. Also, the GPU can operate on multiple data streams simultaneously. For example, some GPUs can execute a fragment shader on up to 24 fragments (pixels) simultaneously, which provides a significant performance advantage over a CPU which can only process one pixel at a time.

Consider a typical image processing application that applies several transforms or operations to a set of image data, stores the result in an IDLgrImage object and then displays the image. In the following figure, the application applies several image operations and creates intermediate images (that may be reused). This process requires a significant amount of computation and data movement before the final image is copied into the image object and the graphic device's texture memory. Additionally, all or most of this process must be repeated any time the parameters of an operation change, reducing interactive performance.

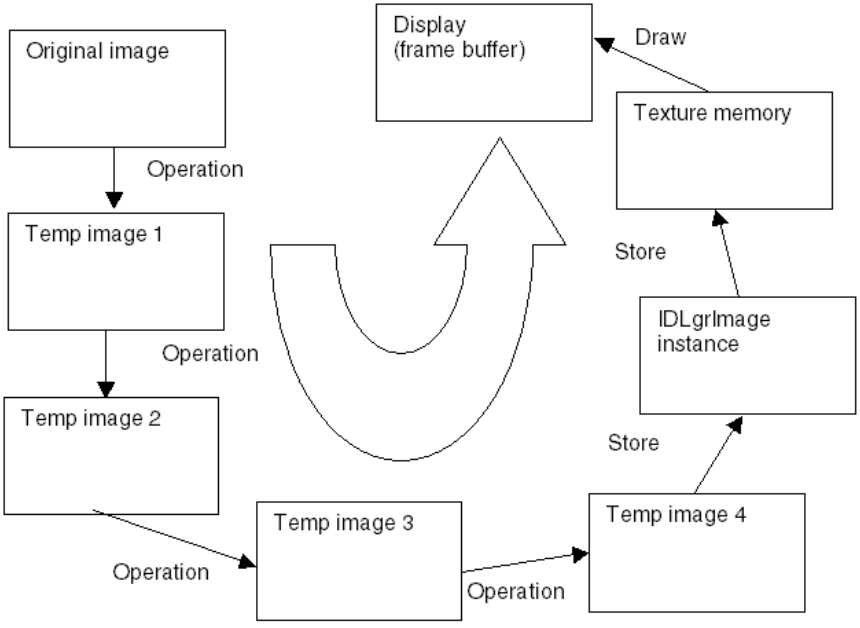


Figure 14-3: Image Processing Pipeline without Shader Program

Consider the same image processing application that uses a shader program to apply the operations. In the following figure, the entire processing cycle is accomplished on the graphics card with the exception of passing in a small amount of data containing operation parameters.

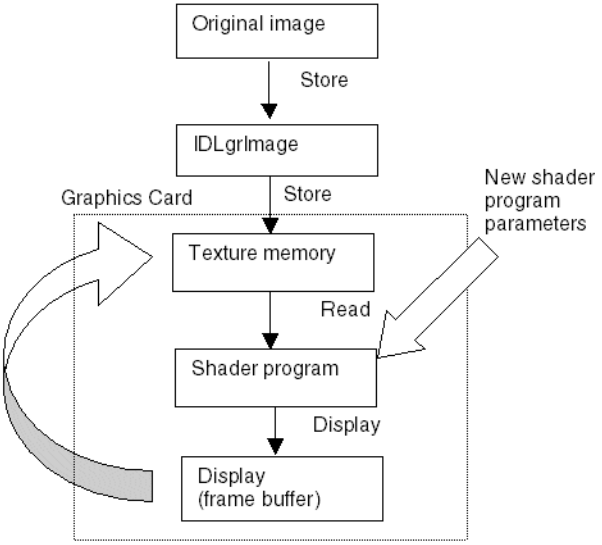


Figure 14-4: Image Processing Pipeline with Shader Program

Without a shader program and suitable hardware, updating an image may require several tenths of a second when the image is large and complex operations are applied. Noticeable display updates may occur with CPU processing. With a shader program, the display rate with the same amount of processing can be hundreds of frames per second. Display updates will be smooth with GPU processing. Display rates of hundreds of frames per second are not always useful, but when lower rates are used, more CPU resources are available for other operations.

Using Shaders in an IDL Application

The IDLgrShader object class exposes OpenGL Shader Language (GLSL) code within an IDL application. Using shader object properties, you can define the required vertex shader and fragment shader components (described in "Vertex and Fragment Shaders" on page 324) by either passing in a string containing the GLSL program or by passing in a filename. Always associate a shader object with an atomic graphic object using the SHADER property. SHADER is a property of the following objects:

IDLgrAxis	IDLgrPlot	IDLgrROIGroup
IDLgrContour	IDLgrPolygon	IDLgrSurface
IDLgrImage	IDLgrPolyline	IDLgrText
IDLgrLight	IDLgrROI	IDLgrVolume

Although a shader object can be associated with any number of the listed graphic objects, a shader program is typically written with a specific object in mind since the IDL application will likely pass object-specific parameters to the shader program. For example, a byte-scale image processing shader would have little applicability to a text object. Additional shader-related properties exist on IDLgrImage, IDLgrLight, IDLgrPlot, IDLgrPolygon, IDLgrPolyline, and IDLgrSurface. These are described in the example sections.

Warning

Setting IDLgrImage RENDER_METHOD=1 (do *not* render image as texture-mapped polygon) disables all shader functionality including the software-based alternative.

Note

In an image processing application, more than a single shader can be associated with an IDLgrImage object through the use of an IDLgrFilterChain object. See "Filter Chain Shaders" on page 355 for details.

Note

Shaders are a hardware-based feature. Be sure to specify the hardware renderer (for example, set the IDLgrWindow RENDERER property to 0).

Display-Only Effects of Shaders

Unless a shader program is associated with an IDLgrImage object, processing results are visible only on the display. The non-display objects (IDLgrBuffer, IDLgrClipboard, IDLgrPrinter and IDLgrVRML) do not support shader functionality.

If the shader is associated with an IDLgrImage object, there are two exceptions to this display-only limitation. You can capture image data after the application of a shader using the IDLgrImage::ReadFilteredData method. See "Capturing Image Data During Shader Execution" on page 335. You can also access full resolution image data if you have implemented a software-based alternative to an image processing shader application. See "Providing a Software Alternative to Shaders" on page 335 for details.

Note

As long as there is hardware support for a shader program, shader program parameters take precedence over any OpenGL fixed pipeline parameters defined using object properties.

Passing Information to a Shader Program

The increased processing power provided by shaders allows the display to be quickly updated when object parameters change. This one-way communication lets you pass in object parameters, such as color updates, but also other variables such as time, for which there is no OpenGL equivalent. The parameter updates cause changes in the color or depth buffers, but no output is returned to the calling application, hence the one-way communication.

Exactly how data is passed to a shader program depends on the target for the parameter data. The two main ways to communicate with a shader program include using "Uniform Variables" described below and "Attribute Variables" on page 332. IDL activates the shader program when the application draws the scene containing the graphic object with the associated IDLgrShader object. IDL passes the uniform variable and/or vertex attribute data that you set with the SetUniformVariable and SetVertexAttributeData methods to the shader program.

Warning

Uniform and attribute variable names are case-sensitive, unlike most variable names in IDL.

Note

If there is insufficient support for the shader program, IDL draws the scene as if there was no shader object present.

Uniform Variables

Uniform variables contain small amounts of data that change infrequently (not more often than when the associated object is drawn). Use the GetUniformVariable and SetUniformVariable methods of the IDLgrShader object to retrieve or pass a named uniform variable to a shader program.

Reserved Uniform Variables

If an IDLgrShader or object subclassing from IDLgrShader is associated with an image, surface or polygon object, IDL sets a number of reserved uniform variables. All reserved uniform variable names begin with "_IDL_".

- **_IDL_ImageStep** — this uniform variable is of GLSL type `vec2`. It contains the values `[1/width, 1/height]`. These values are useful for convolution filters that must locate adjacent texels for convolution kernel computations,

and are used with IDLgrShaderConvol3. This uniform variable is set only when a shader is associated with an IDLgrImage.

- **_IDL_ImageTexture** — this uniform variable is of GLSL type `sampler2D`. When you associate a shader with an IDLgrImage, this variable contains the texture map data associated with the IDLgrImage object. For a shader associated with an IDLgrPolygon or IDLgrSurface, this is the image data that was set using the TEXTURE_MAP property. In the shader program, use the GLSL `texture2D` function to access the texture data.

  Note

  IDL always uses OpenGL's texture unit 0 to store the texture used to draw an IDLgrImage object. IDL also uses texture unit 0 for textures associated with the TEXTURE_MAP property of IDLgrPolygon and IDLgrSurface. If you define a `sampler2D` uniform variable in your shader program and do not initialize it with the SetUniformVariable method in your IDL application, OpenGL associates your `sampler2D` uniform variable with texture unit 0. This automatic association ensures correct operation because your GLSL sampler is referencing the correct texture. This feature may be useful when using shader programs from outside sources. For example if you obtain a shader program from the Internet that performs a type of image filtering, it probably defines a `sampler2D` uniform variable, perhaps named *image*. You can use the shader program without modification and not bother setting the uniform variable called *image* in your IDL code since IDL and OpenGL will correctly associate your IDLgrImage data with the `sampler2D` uniform variable.

  However, it may be good form and improve self-documentation to use the IDL reserved uniform variable, `_IDL_ImageTexture`, to explicitly indicate that the shader program is using the IDL texture as described above.

While the data is automatically associated with these uniform variables and made available to the shader program, you still must define them in the shader program to access the data from within the shader program.

Note

If you are layering multiple textures on a surface or polygon, see "Uniform Variables and Multi-Texture Shaders" on page 370 for information on how to manage reserved and custom uniform variables.

Attribute Variables

Attribute variables contain per-vertex data that is passed to the vertex shader program. This type of data changes frequently (often for each vertex). Modifying object vertices can display movement within a scene. For example, an attribute variable that contains per-vertex velocity vectors multiplied by a uniform variable that contains a time value generates an offset location for each vertex. When this vertex program runs repeatedly with increasing time values, it simulates the motion of a set of vertices where each vertex has its own velocity vector representing its own movement direction. Such a vertex program can be used to visualize the path of moving particles.

Use the GetVertexAttributeData and SetVertexAttributeData methods of the graphic object (not the shader object since vertex data is intimately related to the object vertices) to retrieve or pass a named attribute variable to a shader program. See "Vertex Shaders" on page 359 for an example.

Note

Within a GLSL program, a *varying variable* passes data from the vertex shader to the fragment shader. These variables are defined at each vertex and interpolated across a graphic object to produce a perspective-corrected value at each fragment. This type of variable cannot be directly accessed from IDL.

Library of Pre-built Shader Objects

The IDL distribution includes two shader objects that subclass from IDLgrShader. These pre-built shader objects let you quickly add the functionality of a shader to an image processing application without having to write any GLSL code. These subclasses also provide a software fallback mechanism to apply the image processing step to the image data in the absence of shader hardware. The shader code and the corresponding equivalent IDL code (software fallback) are contained within a single object, which lets you easily add this functionality to an application without worrying about whether or not your user has graphics hardware that supports shaders.

The two pre-built shader objects are:

- **IDLgrShaderBytscl** — highlights features by modifying the input and output levels of the associated IDLgrImage object. See "IDLgrShaderBytscl" (*IDL Reference Guide*) for details and an example.
- **IDLgrShaderConvol3** — defines a convolution filter to smooth, sharpen, perform edge detection, or perform a custom convolution operation when associated with an IDLgrImage object. See "IDLgrShaderConvol3" (*IDL Reference Guide*) for details. See "Filter Chain Shaders" on page 355 for an example.

Image Filter Shaders

Image shader programs are particularly easy to create, for a couple of reasons:

- The IDLgrImage object uses a texture-mapped polygon to draw the image. Most image filters do not change the size or the shape of the image, making it unnecessary to modify the vertices of the polygon. Therefore, a very trivial vertex shader component is all that is required.
- Each image pixel color is going to be completely determined by the image filter calculation, with no lighting or shading effects. Therefore, there is no need to worry about applying lighting and shading calculations in a shader program. This further simplifies the shader program.

There are several ways to incorporate shader functionality into an image processing application. You can either use one of the pre-built shader objects (IDLgrShaderBytscl or IDLgrShaderConvol3) or create a custom shader program. If you design your own shader, you have additional options that include using a IDLgrFilterChain object to link a number of shaders together and apply them successively to the image data. See the following topics for sample applications:

- “Library of Pre-built Shader Objects” on page 333 — provides information on the pre-defined IDLgrShaderBytscl and IDLgrShaderConvol3 objects. These are excellent options if you need byte scaling or convolution filtering functionality, and do not want to write custom GLSL shader programs.
- “Altering RGB Levels Using a Shader” on page 336 — creates a simple shader program that allows you to interactively alter the red, green or blue levels in an RGB image.
- “Applying Lookup Tables Using Shaders” on page 342 — loads a LUT into a one-dimensional image object so that the shader program can efficiently accesses it as a texture map.
- “High Precision Images” on page 349 — shows how to display 16-bit and 11-bit images using the full precision of the data and how to display an 11-bit image with a contrast adjustment LUT.
- “Filter Chain Shaders” on page 355 — lets you apply a sequence of shaders to a single image object.

Providing a Software Alternative to Shaders

When the appropriate graphics hardware support is missing, a software-based alternative can be provided for image processing applications (when the shader program is associated with an IDLgrImage).

When IDL renders the image and hardware shader support is missing, the IDLgrShader::Filter method is *automatically* called when the image is drawn. (You never call this method directly.) In this method, add code that provides a software-based equivalent to the shader program functionality that will be used when there is insufficient hardware support for the shader program. See "Hardware Requirements for Shaders" on page 321 for graphics card requirements.

Note

When developing a software fallback, use the FORCE_FILTER property during shader object initialization to force the system to test the software-based alternative even when sufficient hardware support is available.

If there is no software fallback specified, application execution simply continues as if there were no shader program. Also, no software fallback is available when a shader is associated with a non-image object.

Caching Shader Results

If a shader object is associated with an IDLgrImage object, you may set the IDLgrShader CACHE_RESULT property to determine whether a shader program is executed every time Draw is called. If this property is set to 1, the image is cached after running the shader program and the cached image is used in subsequent draws until shader program parameters are changed. If this property is set to 0 (the default), the result of running the shader program is not cached. See the property description for details on when each CACHE_RESULT setting may prove more useful. If a software fallback is used, the result is always cached.

Capturing Image Data During Shader Execution

When you apply one or more shader programs to image data, you can capture the results of the image filtering shader operation using the IDLgrImage ReadFilteredData method. Using this method, you can capture a portion of a tiled image, capture the image after applying a single shader, or capture the image after applying any number of shaders in a filter chain sequence. See "IDLgrImage::ReadFilteredData" (*IDL Reference Guide*) for details. After reading the data, you can place it in a new image object and print or display the result.

Altering RGB Levels Using a Shader

This shader program example lets you interactively apply color level correction to an image when you view it. This does not modify the image data itself. This example places the original image data in an IDLgrImage object and attaches the custom shader object using the SHADER property. It then creates a simple user interface that lets you alter the color levels and passes these values to the shader program in a named uniform variable. The Filter method implements the software fallback. When the correct graphics hardware is unavailable, IDL automatically calls the Filter method.

Example Code

See `shader_rgb_doc__define.pro`, located in the `examples/doc/shaders` subdirectory of the IDL distribution, for the complete, working example. Run the example by creating an instance of the object at the IDL command prompt using `orgbshader=OBJ_NEW('shader_rgb_doc')` or view the file in an IDL Editor window by entering `.EDIT shader_rgb_doc__define.pro`.

The example code differs slightly from that presented here for the sake of clarity. Whereas the working example includes code needed to support user interface interaction, the following sections leave out such modifications to highlight the shader program components.

Basic RGB Shader Object Class

First, create a basic object class that inherits from IDLgrShader:

```
; Initialize object.
FUNCTION shader_rgb_doc::Init, _EXTRA=_extra
    IF NOT self->IDLgrShader::Init(_EXTRA=_extra) THEN $
        RETURN, 0
    RETURN, 1
END

; Clean up.
PRO shader_rgb_doc::Cleanup
    self->IDLgrShader::Cleanup
END

; Filter method for software fallback option.
FUNCTION shader_rgb_doc::Filter, Image
   RETURN, Image
END
```

```
; Class definition.
PRO shader_rgb_doc__define
COMPILE_OPT hidden
    struct = { shader_rgb_doc, $
               INHERITS IDLgrShader $
             }
END
```

Uniform Variable for RGB Values

In this example, a uniform variable contains the values of the red, green and blue levels. You can set or change uniform variables anytime before you draw the scene and their values will remain in effect until you change them again. These types of variables are perfect for making minor adjustments to the image filter and then viewing the image to see if the result is satisfactory.

First set the uniform variable to a reasonable default value such as [1,1,1] before you start, otherwise the shader program defaults of [0,0,0] will make the image look dim. Add the following line to your Init function:

```
self->SetUniformVariable, 'scl', [1.0, 1.0, 1.0]
```

Warning

The uniform variable name is case-sensitive, unlike most variable names in IDL.

This example lets you change color levels using sliders. You can read the slider values from your GUI, and modify the uniform variable at any time. Assuming that the instance of your `shader_rgb_doc` object is called *oShaderRGB* and *red*, *green* and *blue* are floating point values, update the value of the uniform variable as follows:

```
oShaderRGB->SetUniformVariable, 'scl', [red, green, blue]
```

Once the needed elements are defined, associate your shader object with *oImage*, an image object (that has been previously defined).

```
oImage->SetProperty, SHADER=self
```

Once the shader object is associated with the image, shader program display updates are activated any time the SetUniformVariable method is called.

Software Fallback for RGB Shader

IDL calls the Filter method when shader functionality is not supported by the graphics hardware. Providing a software-based fallback is never a requirement and you may choose not to if you know sufficient hardware will always be available.

However, it is good practice to write this method just in case the application is ever executed on a machine without suitable hardware.

In the Filter method, retrieve the uniform variable values using GetUniformVariable, and then return a modified copy of the image data.

```
Function shader_rgb_doc::Filter, Image

newImage=Image
self->GetUniformVariable, 'scl', s
newImage[0,*,*] *= s[0]
newImage[1,*,*] *= s[1]
newImage[2,*,*] *= s[2]

RETURN, newImage

END
```

IDL always passes the image to the Filter method in RGBA floating-point pixel-interleaved format, so you don't have to worry about a lot of input data combinations. IDL also clamps the data this function returns to the [0.0, 1.0] range and scales it to the correct pixel range, usually [0, 255], for your display device.

Note

Uniform variables are, in a sense, free-form properties in the IDLgrShader superclass. Within the Filter method, accessing the scale vector from the uniform variable maintains consistency since this is same place the hardware shader obtains it. This reduces the chance for confusion.

At this point, you can test your work by writing a simple display program that loads your data into an IDLgrImage object, creates an instance of your `shader_rgb_doc` object, and attaches the filter to your image object by setting the object reference of the shader in the SHADER property of IDLgrImage. You also need to set the FORCE_FILTER property on class initialization so that the filter fallback runs, even if you have shader hardware. You can force use of the fallback either when creating the shader object:

```
oShaderRGB = OBJ_NEW('shader_rgb_doc', /FORCE_FILTER)
```

or explicitly in the shader object's Init method:

```
FUNCTION shader_rgb_doc::Init, _EXTRA=_extra

IF NOT self->IDLgrShader::Init(_EXTRA=_extra, /FORCE_FILTER) $
   THEN $
      RETURN, 0
...
```

Hardware Shader Program for RGB Shader

The OpenGL Shading Language (GLSL) is a vast subject that requires extensive study to develop an expert level of programming, a subject that is impossible to cover here. However, this example is relatively simple and you can likely easily follow along with the code required for the vertex and fragment shader portions of the shader program. All shader programs need a vertex program and a fragment program.

RGB Vertex Shader Program

The following vertex shader is fairly common among image filtering shader programs. Add the following code to the bottom of your Init function:

```
vertexProgram = $
[ $
'void main (void) {', $
   '  gl_TexCoord[0] = gl_MultiTexCoord0;', $
   '  gl_Position = ftransform();', $
'}' ]
```

The first line after `main()` transfers the texture coordinate from OpenGL's `Texture Unit 0` into the current texture coordinate predefined variable. Remember that IDL draws its images with texture maps applied to rectangles, so you need to pass along the texture coordinate. *IDL always uses Texture Unit 0 when drawing images*. The `gl_TexCoord[0]` is a varying variable that transmits data from the vertex program to the fragment shader program.

The next line in the program simply applies the current OpenGL `ModelViewProjection transform` to the vertex, so that it ends up in the right place on the screen.

RGB Fragment Shader Program

The fragment program of an image filtering shader program is where all the work happens. Add the following to the Init function as well:

```
fragmentProgram = $
[ $
'uniform sampler2D _IDL_ImageTexture;', $
'uniform vec3 scl;', $
'void main(void) {', $
   'vec4 c = texture2D(_IDL_ImageTexture, gl_TexCoord[0].xy);', $
   '    c.rgb *= scl;', $
   '    gl_FragColor = c;', $
'}' ]
```

This GLSL code can be translated relatively easily. IDLgrImage uses textures to draw image data. Access the texture map associated with the base image's data in the IDL reserved uniform variable, `_IDL_ImageTexture`, which is automatically created for the base image. The sixth line in the program above fetches the image pixel (a texture texel) from the image texture and stores it in *c*, which is a 4-element vector that represents the RGBA channel data. Modify the color of the texel in the next tile using the uniform variable, *scl*, declared on line four. Finally, tell OpenGL about the new color for this particular pixel on the screen by setting `gl_FragColor`. OpenGL clamps the pixel color values to the appropriate range for your display.

This fragment program runs once for every pixel (fragment) on your screen that is covered by the image.

Assign RGB Shader Program to Shader Object

You need to supply the program code to the shader object so that it is available to the graphics card when it is needed. To accomplish this, you can use shader object properties VERTEX_PROGRAM_STRING and FRAGMENT_PROGRAM_STRING to associate inline shader program components with the shader object.

Note

With more complicated (longer) shader programs, it may be easier to keep the shader program components in separate files. In such a case, associate the shader program elements with a shader object using the VERTEX_PROGRAM_FILE and FRAGMENT_PROGRAM_FILE properties.

Add the following code to the bottom of your Init function.

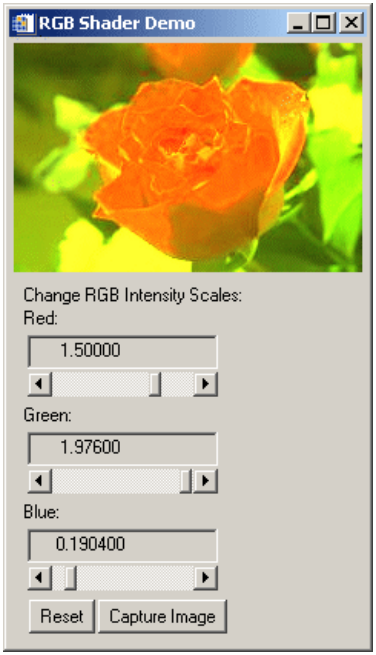
```
self->IDLgrShader::SetProperty, $
  VERTEX_PROGRAM_STRING=STRJOIN(vertexProgram, STRING(10B)), $
  FRAGMENT_PROGRAM_STRING=STRJOIN(fragmentProgram, STRING(10B))
```

Add newlines (`STRING(10B)`) so that the shader program compiler sees your program as a single long string containing many source code lines, instead of one long line. If you ever get a compile-time error, the shader compiler can tell you on what line the error occurred when you insert the newlines.

Tip

Remove the FORCE_FILTER keyword from the initialization function if you have been testing your software fallback.

The following image shows the result of modifying the RGB levels of an image of a rose.



Applying Lookup Tables Using Shaders

The IMAGE_1D property on the IDLgrImage object lets you load color lookup table (LUT) values into a texture map and pass the LUT to a shader program. LUTs are useful for a number of tasks including:

- Displaying palletized images.
- Adding color to greyscale images.
- Optimizing the evaluation of expensive functions. For example, if your image is 8-bit greyscale and you need to apply an expensive function to each pixel it is normally more efficient to pass each of the 256 greyscale values to the function and store the result in a 256 entry LUT used for drawing.
- Adjusting image brightness, gamma, contrast, color balance and other settings.
- Adjusting data ranges such as converting an 11-bit image to 8-bits for display (see “High Precision Images” on page 349 for more information).

Example Code

See `shader_lut_doc__define.pro`, located in the `examples/doc/shaders` subdirectory of the IDL distribution, for the complete, working example. Run the example by creating an instance of the object at the IDL command prompt using `oLUTshader=OBJ_NEW('shader_lut_doc')` or view the file in an IDL Editor window by entering `.EDIT shader_lut_doc__define.pro`.

The example code differs slightly from that presented here for the sake of clarity. Whereas the working example includes code needed to support user interface interaction, the following sections leave out such modifications to highlight the shader program components.

Basic LUT Shader Object Class

The `shader_lut_doc` object class inherits from IDLgrShader and contains the Filter method, just like the “Basic RGB Shader Object Class” on page 336. See that section for the base code or the example for the complete code. The one difference is this example uses the shader object VERTEX_PROGRAM_FILENAME and FRAGMENT_PROGRAM_FILENAME properties, which reference external shader program files for the vertex and fragment shader components.

Uniform Variable for LUT Example

In this example, a uniform variable named *lut* contains the values of the 256-element array of color table values. This can either be a custom LUT such as an enhanced greyscale color table, or one of the predefined IDL LUTs.

The following code creates a greyscale LUT defined by a curve rather than a linear ramp, making the dark areas darker and the light areas lighter. Notice that the 256-entry LUT is loaded into a one-dimensional image (an IDLgrImage object with IMAGE_1D property set). This IDLgrImage is automatically converted into a texture map for use by the shader. SetUniformVariable is called with the name of the uniform variable and the value (the image object) so the shader can access the texture map containing the LUT.

```
; Create enhanced grayscale LUT and store in 1-D IDLgrImage.
x = 2*!PI/256 * FINDGEN(256) ;; 0 to 2 pi
lut = BYTE(BINDGEN(256) - sin(x)*30) ;; Create 256 entry

oLUT = OBJ_NEW('IDLgrImage', lut, /IMAGE_1D)

; Store LUT in uniform variable named lut.
self->SetUniformVariable, 'lut', oLUT
```

Warning

The uniform variable name is case-sensitive, unlike most variable names in IDL.

The LUT is loaded into a texture map instead of a uniform variable array because it is more efficient to load and index the LUT when it is in a texture. In addition, under certain circumstances you can use bilinear filtering to interpolate between values in the LUT if it is in a texture map.

A side effect of using a texture map is it is limited by the maximum texture size (MAX_TEXTURE_DIMENSIONS in IDLgrWindow::GetDeviceInfo). On most hardware today this is 4096 by 4096 pixels, so if your LUT is larger than this you will need to work around this limitation (using a 2-D texture map is one possible solution). Also, as texture maps must be a power of 2 in size (128, 256, 512, 1024, etc.), ensure the size of your LUT is a power of 2 to keep it from being scaled to the next higher power of 2.

To display palletized images or to add color to greyscale images, simply load an RGB LUT into the 1D IDLgrImage rather than a greyscale LUT. The shader code remains exactly the same. (The `shader_lut_doc__define.pro` program lets you apply either the enhanced greyscale or one of IDL's pre-defined colortables.)

Hardware Shader Program for LUT Shader

This example reads the shader source from text files. The vertex shader (`LUTShaderVert.txt` located in `examples/doc/shaders`) contains the following code:

```
void main (void)
{
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position = ftransform();
}
```

This basic vertex program passes along the texture coordinate and then applies a transform to the vertex to correctly position it on the screen. The `gl_TexCoord[0]` is a varying variable that transmits data from the vertex program to the fragment shader program.

The fragment shader (`LUTShaderFrag.txt` located in `examples/doc/shaders`) contains the following code:

```
uniform sampler2D _IDL_ImageTexture;
uniform sampler1D lut;

void main(void)
{
    float i = texture2D(_IDL_ImageTexture, gl_TexCoord[0].xy).r;
    gl_FragColor = texture1D(lut, i);
}
```

The fragment shader is where the lookup happens. The uniform variable, *lut*, which was defined in the IDL application using SetUniformVariable, contains the lookup table in a 1-D texture (of GLSL type `sampler1D`). As previously explained, the LUT is loaded into a texture map for efficiency.

The `_IDL_ImageTexture` variable is a reserved uniform variable that provides access to the 2-D base image (of GLSL type `sampler2D`). When a shader object is associated with an IDLgrImage object, and the uniform variable is not defined using SetUniformVariable in the IDL application, the base image object (a texture mapped onto a rectangle) is stored in a reserved uniform variable named `_IDL_ImageTexture`. The base image is the IDLgrImage to which the shader is attached. If it is attached to more than one image, the base image is the one currently being shaded. Non-base images are those passed to the shader program using SetUniformVariable.

Since more than one texture is used in the rendering of the image (the `_IDL_ImageTexture` base image texture and the *lut* texture), this is referred to as multi-texturing.

The GLSL `texture2D` procedure call reads the texel at the current texture coordinate. This procedure typically returns a floating point, four-element vector (containing red, green, blue and alpha values). But with a greyscale image, the red, green, and blue values are the same, so the appending `.r` keeps only the red channel and assigns it to the float *i*.

The GLSL `texture1D` procedure takes two parameters, the *lut* and *i* (the texture coordinate that instructs it which texel to sample). This value normally ranges from 0.0 to 1.0 (0.0 being the first texel, 1.0 the last). Since the value read from the image into *i* also normally ranges between 0.0 and 1.0, it is possible to use it directly as a texture coordinate to do the lookup.

When performing a lookup on the CPU, you directly access the LUT array using the pixel value as the index. A pixel value of 0 corresponds to the first entry in the LUT and a pixel value of 255 corresponds to the last entry.

However, in a shader program the texture coordinate lookup is possible because before a pixel reaches the fragment shader it is converted to floating point by OpenGL. In the case of an 8-bit greyscale image, the range is 0.0 to 1.0. That means a pixel with value 0 becomes 0.0 and 255 becomes 1.0. When doing the coordinate texture lookup on the GPU, the `texture1D` procedure does the lookup by using the converted pixel values where pixel value of 0 corresponds to the first LUT entry and a pixel value of 1.0 (converted from 255) corresponds to the last entry.

Assign LUT Shader Program to Shader Object

You need to supply the program code to the shader object so that it is available to the graphics card when it is needed. To accomplish this, you can use shader object properties VERTEX_PROGRAM_FILE and FRAGMENT_PROGRAM_FILE to associate external shader program components with the shader object.

Add the following code to the bottom of your Init function:

```
vertexFile=filepath('LUTShaderVert.txt', $
   SUBDIRECTORY=['examples','doc', 'shaders'])
fragmentFile=filepath('LUTShaderFrag.txt', $
   SUBDIRECTORY=['examples','doc', 'shaders'])

self->IDLgrShader::SetProperty, $
   VERTEX_PROGRAM_FILENAME=vertexFile, $
   FRAGMENT_PROGRAM_FILENAME=fragmentFile
```

At this point, you can easily add image display code to your program and test your LUT shader. The result of applying one of IDL's pre-defined colortables appears in the following figure.

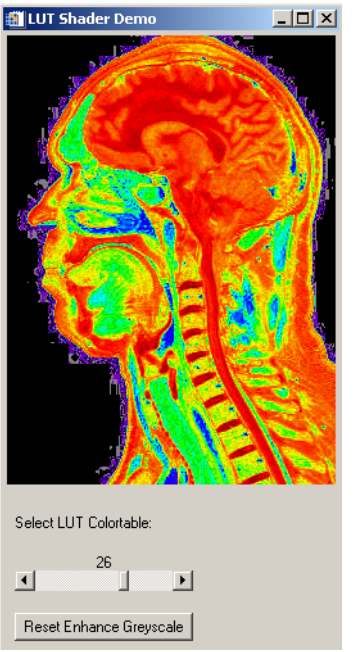


Figure 14-5: LUT Shader Example

Software Fallback for the LUT Shader

The following code performs the LUT lookup. When there is not sufficient hardware support for shaders or when the FORCE_FILTER keyword is set on initialization, the colortables changes result from the following code instead of a shader program. You will likely find that performance slows significantly.

```
Function shader_lut_doc::Filter, Image

; Allocate return array of same dimension and type.
sz = SIZE(Image)
newImage = FLTARR(sz[1:3], /NOZERO)

; Get the LUT uniform variable.
self->GetUniformVariable, 'lut', oLUT
```

```
; Read the LUT data from the 1-D image.
oLUT->GetProperty, DATA=lut
FOR y=0, sz[3]-1 DO BEGIN
    FOR x=0, sz[2]-1 DO BEGIN
        ; Read from the image.
        idr = Image[0,x,y]
        ; Convert from 0.0-1.0 back to 0-255.
        idr *= 255

        ; Get the number of image channels.
        szlut = SIZE(lut)
        IF szlut[0] EQ 1 THEN BEGIN
            ; Greyscale LUT, only 1 channel.
            grey = lut[idr]
            fgrey = FLOAT(grey) / 255.0
            newImage[0,x,y] = fgrey
            newImage[1,x,y] = fgrey
            newImage[2,x,y] = fgrey
            newImage[3,x,y] = 1.0
        ENDIF ELSE BEGIN
            ;; RGB LUT.
            rgb = lut[*, idr]
            frgb = FLOAT(rgb) / 255.0
            newImage[0:2,x,y] = frgb
            newImage[3,x,y] = 1.0
        ENDELSE
    ENDFOR
ENDFOR
RETURN, newImage
END
```

IDL always passes the image to the Filter method in RGBA floating-point pixel-interleaved format, so you don't have to worry about a lot of input data combinations. IDL also clamps the data this function returns to the [0.0, 1.0] range and scales it to the correct pixel range, usually [0, 255], for your display device.

Note

Uniform variables are, in a sense, free-form properties in the IDLgrShader superclass. Within the Filter method, accessing the *lut* texture map from the uniform variable maintains consistency since this is same place the hardware shader obtains it. This reduces the chance for confusion.

At this point, you can test your work by writing a simple display program that loads your data into an IDLgrImage object, creates an instance of your `shader_lut_doc` object and attaches the LUT to your image object by setting the object reference of the shader in the SHADER property of IDLgrImage. You also need to set the

FORCE_FILTER property on class initialization so that the filter fallback runs, even if you have shader hardware:

```
oLUTshader = OBJ_NEW('shader_lut_doc', /FORCE_FILTER)
```

High Precision Images

Traditionally, most computer graphics cards and monitors have been able to display a maximum color depth of 8-bits per channel. In such cases high-precision images that exceed 8-bits per channel must be converted to 8-bit for display. This was traditionally accomplished using the IDL BYTSCL function, which was limited by the processing capabilities of the CPU. Fortunately, this conversion can now be accomplished by the GPU. The full precision image is passed to the video card memory once and is then converted as it is rendered.

OpenGL Conversion of Image Data to Texture Data

It is important to understand how OpenGL converts a high precision image to a texture map before writing a shader program. The graphics card vendor ultimately decides what formats are supported. Using the IDLgrImage INTERNAL_DATA_TYPE property, you tell OpenGL in what format you would like the texture stored. The following table describes the relationship between OpenGL types and the INTERNAL_DATA_TYPE property value.

OpenGL	INTERNAL_DATA_TYPE Setting	Description
RGBA8	1	8-bit unsigned bytes per channel, widely supported
RGBA16F	2	16-bit floating point with 1 sign bit, 5 exponent bits and 10 mantissa bits
RGBA32F	3	32-bit floating point, which is standard IEEE float format

Table 14-1: Texture Data Types and Settings

An IDLgrImage will accept data of type BYTE, UINT, INT and FLOAT. When the texture map is created the data from IDLgrImage is converted to the type specified in INTERNAL_DATA_TYPE.

Note

If your image data is floating point, your fragment shader must scale it to the range 0.0 to 1.0 before writing it to `gl_FragColor` or you need to scale it to the range of 0.0 to 1.0 before setting it on the IDlgrImage.

If INTERNAL_DATA_TYPE is set to floating point (INTERNAL_DATA_TYPE equals 2 or 3), image data conversion is performed by OpenGL as follows where *c* is the color component being converted:

Image Data Type	Floating Point Conversion
BYTE	$c/(2^8-1)$
UINT	$c/(2^{16}-1)$
INT	$(2c+1)/(2^{16}-1)$
FLOAT	c

Table 14-2: OpenGL Conversion of Image Data to Floating Point

If INTERNAL_DATA_TYPE is 1 (8-bit unsigned byte), then the image data is scaled to unsigned byte. This is equivalent to a linear BYTSCL from the entire type range (e.g. 0-65535) to unsigned byte (0-255).

Note

INTERNAL_DATA_TYPE of 0, the default, maintains backwards compatibility by converting the image data to byte without scaling.

To avoid data loss during conversion, you should choose an internal data type with sufficient precision to hold your image data. For example, with a 16-bit UINT image that uses the full range of 0-65535, if you set INTERNAL_DATA_TYPE to 2 (16-bit floating point), your image will still be converted to the range of 0.0 to 1.0, but some precision will be lost (due to the mantissa of a 16-bit float being only 10 bits). If you need a higher level of precision, set INTERNAL_DATA_TYPE to 3 (32-bit floating point). However, on some cards there may be a performance penalty associated with the higher level of precision, and requesting 32-bit floating point will certainly require more memory.

Once the image has been converted to a texture map it can be sampled by the shader. The GLSL procedure, `texture2D`, returns the sampled texel in floating point (0.0 to 1.0). Therefore, if the INTERNAL_DATA_TYPE is 1 (unsigned byte) the texel is converted to floating point, using $c/(2^8 - 1)$, before being returned.

Examples of Handling High-Precision Images

The following examples provide guidelines for reading in various types of image data including how to set the IDLgrImage INTERNAL_DATA_TYPE property and supporting fragment shader code. However, due to the size limitations of the IDL distribution, high-precision images are not included, so you will need to use your own data to create working examples. See the following sections:

- "Displaying a 16-bit UINT Image" below
- "Displaying an 11-bit UINT Image" on page 352
- "Displaying an 11-bit UINT Image with Contrast Adjustment" on page 353

Displaying a 16-bit UINT Image

In this example, the input image (*uiImageData*) is 16-bit unsigned integer greyscale image that uses the full range of 0 to 65535. The goal is to display the entire range using a linear byte scale. Traditionally we'd use the BYTSCL function in IDL prior to loading data into the IDLgrImage object:

```
ScaledImData = BYTSCL(uiImageData, MIN=0, MAX=65535)
oImage = OBJ_NEW('IDLgrImage', ScaledImData, /GREYSCALE)
```

To have the GPU do the scaling, load the unscaled image data into the IDLgrImage and set INTERNAL_DATA_TYPE to 3 (32-bit floating point):

```
oImage = OBJ_NEW('IDLgrImage', uiImageData, $
    INTERNAL_DATA_TYPE=3, /GREYSCALE, SHADER=oShader)
```

The fragment shader is extremely simple. Here, the reserved uniform variable, `_IDL_ImageTexture`, represents the base image in IDL:

```
uniform sampler2D _IDL_ImageTexture;

void main(void)
{
gl_FragColor = texture2D(_IDL_ImageTexture, gl_TexCoord[0].xy);
}
```

All we are doing is reading the texel with `texture2D` and setting it in `gl_FragColor`. You will notice that there is no explicit conversion to byte because this is handled by OpenGL. The value written into `gl_FragColor` is a GLSL type `vec4` (4 floating point values, RGBA). OpenGL clamps each floating point value to the range 0.0 to 1.0 and converts it to unsigned byte where 0.0 maps to 0 and 1.0 maps to 255. So all we have to do is read the texel value from `_IDL_ImageTexture` and set it into `gl_FragColor`.

Displaying an 11-bit UINT Image

An 11-bit unsigned integer image is usually stored in a 16-bit UINT array, but with only 2048 (2^{11}) values used. For this example, let's say the minimum value is 0 and the max is 2047. Traditionally this would be converted to byte as follows:

```
ScaledImData = BYTSCL(uiImageData, MIN=0, MAX=2047)
oImage = OBJ_NEW('IDLgrImage', ScaledImData, /GREYSCALE)
```

To scale on the GPU we again load the image with the original data. This time INTERNAL_DATA_TYPE can be set to 2 (16-bit float) as this can hold 11-bit unsigned integer data without loss of precision:

```
oImage = OBJ_NEW('IDLgrImage', uiImageData, $
   INTERNAL_DATA_TYPE=2, /GREYSCALE, SHADER=oShader)
```

The fragment shader looks like the following where `_IDL_ImageTexture` represents the base image in IDL:

```
uniform sampler2D _IDL_ImageTexture;

void main(void)
{
gl_FragColor = texture2D(_IDL_ImageTexture, gl_TexCoord[0].xy) *
   (65535.0 / 2047.0);
}
```

The only difference between this 11-bit example and the previous 16-bit example is the scaling of each texel. When the 16-bit UINT image is converted to floating point, the equation $c/(2^{16} - 1)$ is used (see Table 14-2) so 65535 maps to 1.0. However, the maximum value in the 11-bit image is 2047, which is 0.031235 when converted to floating point. This needs scaled to 1.0 before being assigned to `gl_FragColor` if we want 2047 (image maximum) to map to 255 (maximum intensity) when the byte conversion is done. (Remember a value of 1.0 in `gl_FragColor` is mapped to 255.)

It's possible to implement the full byte scale functionality on the GPU, and let the user interactively specify the input min/max range by passing them as uniform variables. There is a performance advantage to doing this on the GPU as the image data only needs to be loaded once and the byte scale parameters are changed simply by modifying uniform variables. See "IDLgrShaderBytscl" (*IDL Reference Guide*) and the associated example to see how this can be achieved.

Displaying an 11-bit UINT Image with Contrast Adjustment

The previous example applied a linear scaling to the 11-bit data to convert it to 8-bit for display purposes. Sometimes it's useful to apply a non-linear function when converting to 8-bit to perform contrast adjustments to compensate for the non-linear response of the display device (monitor, LCD, projector. etc.).

For an 11-bit image this can be achieved using a LUT with 2048 entries where each entry contains an 8-bit value. This is sometimes referred to as an 11-bit in, 8-bit out LUT, which uses an 11-bit value to index the LUT and returns an 8-bit value.

This is relatively simple to implement on the GPU. First create the 2048 entry contrast enhancement LUT and load it into an IDLgrImage which will be passed to the shader as a texture map (see "Applying Lookup Tables Using Shaders" on page 342 for more information).

```
x = 2*!PI/256 * FINDGEN(256) ;; 0 to 2 pi
lut = BYTE(BINDGEN(256) - sin(x)*30)
; Stretch to 2048 entry LUT.
lut = CONGRID(lut, 2048)
oLUT = OBJ_NEW('IDLgrImage', lut, /IMAGE_1D, /GREYSCALE)
oShader->SetUniformVariable, 'lut', oLUT
```

The image is created as before:

```
oImage = OBJ_NEW('IDLgrImage', uiImageData, $
   INTERNAL_DATA_TYPE=2, /GREYSCALE, SHADER=oShader)
```

The fragment shader looks like the following where `_IDL_ImageTexture` represents the base image in IDL and *lut* is the lookup table.:

```
uniform sampler2D _IDL_ImageTexture;
uniform sampler1D lut;

void main(void)
{
    float i = texture2D(_IDL_ImageTexture, gl_TexCoord[0].xy).r *
            (65535.0/2048.0);
     gl_FragColor = texture1D(lut, i);
}
```

As you can see the texel value is scaled before being used as an index into the LUT.

The following figure shows how the 11-bit to 8-bit LUT is indexed. Only a fraction of the input data range is used (0-2047 out of a possible 0-65535). As 2047 (0.0312 when converted to float) is the maximum value, this should index to the top entry in the LUT. So we need to scale it to 1.0 by multiplying by 32.015. Now the range of values in the image (0-2047) index the entire range of entries in the LUT.

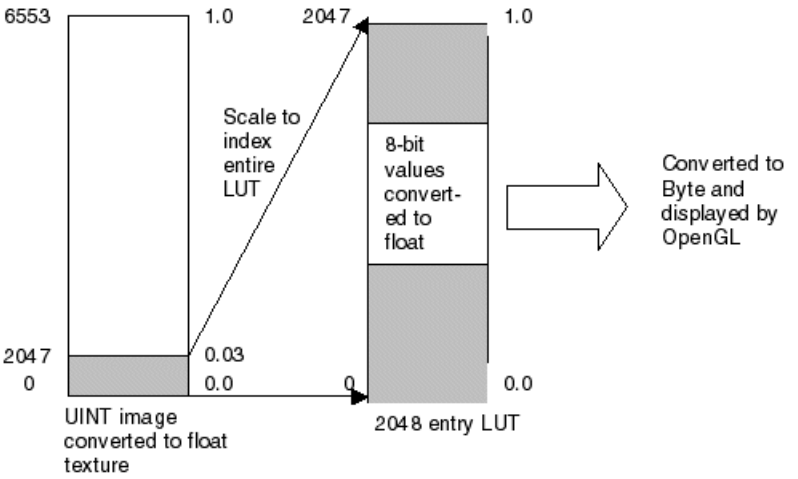


Figure 14-6: Conversion to BYTE Using 11-bit to 8-bit LUT

Although this could be done on the CPU, it is much more efficient to do it on the GPU since the image data only needs to be loaded once and the display compensation curve can be modified simply by changing data in the IDLgrImage holding the LUT.

Filter Chain Shaders

A series of image filtering shaders can be grouped and applied sequentially to an image. To do so, add each shader to an IDLgrFilterChain object, which is a specialized type of container designed to hold IDLgrShader objects or objects that subclass from IDLgrShader. Then associate the IDLgrFilterChain object with the IDLgrImage object using the SHADER property.

When the scene is drawn, the image data is modified by each shader program according to container order. The output from the first shader is processed by each subsequent shader until all shader programs have been applied. IDL then draws the result to the window.

Note

This functionality requires support for GLSL frame buffer object extension in addition to the standard hardware support required by IDLgrShader. See the IDLgrWindow::GetDeviceInfo methods's FRAMEBUFFER_OBJECT_EXTENSION keyword for details).

The following example creates an IDLgrFilterChain object and lets you add and remove individual IDLgrShaderConvol3 objects, which provide the ability to apply sharpening, smoothing, and edge detection convolution filters to an image. Like the IDLgrShaderBytscl object, the pre-defined IDLgrShaderConvol3 object includes a software fallback that is automatically used when there is not sufficient hardware support for shader operations.

Example Code

See `shader_filterchain_doc__define.pro`, located in the `examples/doc/shaders` subdirectory of the IDL distribution, for the complete, working example. Run the example by entering `obj=OBJ_NEW('shader_filterchain_doc')` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT shader_filterchain_doc__define.pro`.

Basic Filter Chain Shader Object Class

The `shader_filterchain_doc` object class inherits from IDLgrFilterChain, which inherits container manipulation methods from IDL_Container, but also includes the FORCE_FILTER method common to IDLgrShader. This means that you can test any software based alternative code provided in a shader's Filter method as described in "Providing a Software Alternative to Shaders" on page 335. Since this

example uses the pre-defined IDLgrShaderConvol3 object, there is no need to specify vertex or fragment programs since these are inherent to the object definition.

In addition to the typical object definition code, this example creates instances of the four types of pre-defined convolution filters and stores them in an object array:

```
oIdentity = OBJ_NEW("IDLgrShaderConvol3", KERNEL=0)
oSmooth = OBJ_NEW("IDLgrShaderConvol3", KERNEL=1)
oSharpen = OBJ_NEW("IDLgrShaderConvol3", KERNEL=2)
oEdge = OBJ_NEW("IDLgrShaderConvol3", KERNEL=3)
objarray = [oIdentity, oSmooth, oSharpen, oEdge]
```

Since an unmodified image is loaded first, make sure the identity convolution filter is the only item in the filter chain object (*self* in the following lines). Then associate the IDLgrFilterChain object with the image using the SHADER property.

```
self->Add, oIdentity
oImage->SetProperty, SHADER=self
```

In this program, you can select among four check boxes to apply varying combinations of convolution filters to a grayscale image. Each time you select a different check box, the list of shaders are removed from the IDLgrFilterChain container and then the selected items are re-added.

```
; Remove all items from the collection and add back
; selected shaders.
self->Remove, /ALL
selected = WHERE (value EQ 1)
IF N_ELEMENTS(selected) GT 1 || selected NE -1 THEN BEGIN
   self->Add, (*pstate).objarray[where (value EQ 1)]
ENDIF

; Update base and covolution factors for all
; selected shaders.
shaderObjs=self->Get(/ALL, COUNT=count)
FOR i =0, count-1 DO BEGIN
   shaderObjs[i]->SetProperty,
      BASE_BLEND_FACTOR=(*pState).basefactor, $
      CONVOL_BLEND_FACTOR=(*pState).convolfactor
ENDFOR

; Draw.
(*pState).oWindow->Draw, (*pState).oView
```

Note

Shaders are applied in container order. You could use different user interface controls to provide a way to apply shaders in a specific order instead of using check boxes.

Uniform Variables for Filter Chain Example

There are no exposed uniform variables since the IDLgrShaderConvol3 object exposes its uniform variables as properties. When you call SetProperty for one of the convolution shader's properties, it calls SetUniformVariable internally. See "IDLgrShaderConvol3 Properties" (*IDL Reference Guide*) for information on the BASE_BLEND_FACTOR, CONVOL_BLEND_FACTOR, and KERNEL properties. If you look in `idlgrshaderconvol3__define.pro` located in the `lib` subdirectory of the IDL distribution, you will see the following uniform variables in the fragment shader program:

```
uniform sampler2D _IDL_ImageTexture
uniform float BaseBlend
uniform float ConvolBlend
uniform vec2 _IDL_ImageStep
uniform vec4 kernel[9]
```

The `BaseBlend`, `ConvolBlend` and `kernel` variables relate to object properties. The `_IDL_ImageTexture` refers to the base IDLgrImage object and `_IDL_ImageStep` is used by the convolution filtering operation. Both are reserved uniform variables (see "Reserved Uniform Variables" on page 330 for details).

Hardware Shader Program for Filter Chain Example

The fragment and vertex shader programs are incorporated into the IDLgrShaderConvol3 object definition file, `idlgrshaderconvol3__define.pro`, located in the `lib` subdirectory of the IDL distribution.

Software Fallback for the Filter Chain Shader

The IDLgrShaderConvol3 object definition file includes a software fallback option that can be exercised using the IDLgrFilterChain FORCE_FILTER property. Set the property either on object creation or in the FilterChain object's Init method. For example, to use the `shader_filterchain_doc` example with the software fallback, create the object as follows:

```
obj=OBJ_NEW('shader_filterchain_doc', /FORCE_FILTER)
```

Switching the value of the FORCE_FILTER property between 0 and 1 in this example allows you to see the execution speed differences between the hardware and software versions of the filter chain.

When you create a `shader_filterchain_doc` object and select `mineral.png`, you can apply one or more convolution shaders and modify shader parameters as shown in the following figure.

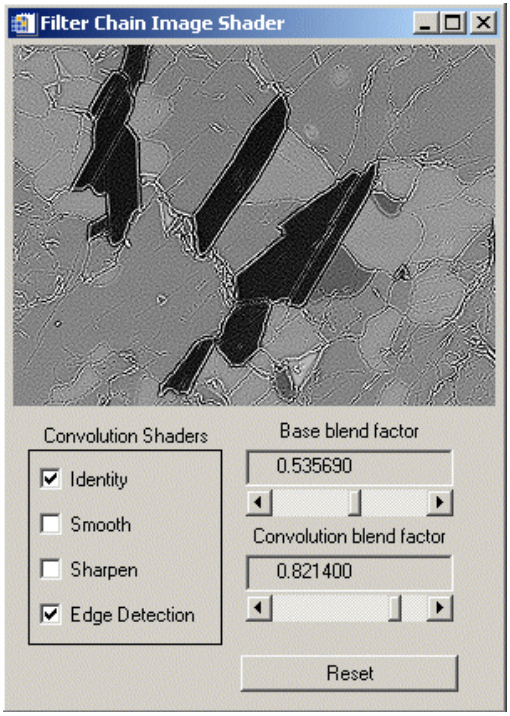


Figure 14-7: Applying a Combination of Shaders Using IDLgrFilterChain

Vertex Shaders

Vertex attribute data may be associated with IDLgrPlot, IDLgrPolygon, IDLgrPolyline, and IDLgrSurface objects using the GetVertexAttributeData and SetVertexAttributeData. While uniform variables are useful for passing concise control parameters to a shader program, they are not suited for passing large amounts of data where the data might contain values that are associated with each vertex. The ability to pass frequently changing per-vertex attribute data to a shader program in an attribute variable lets you show movement within a display (as described in "Attribute Variables" on page 332).

The following example uses an attribute variable to replicate the effect of wind on a set of particles. Each particle has an initial position and a velocity assigned to it. The initial position of the particle can be easily represented by the already-familiar vertex [*x*, *y*, *z*] information.

Example Code

See `shader_vertexwinds_doc.pro`, located in the `examples/doc/shaders` subdirectory of the IDL distribution, for the complete, working example. Run the example procedure by entering `shader_vertexwinds_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT shader_vertexwinds_doc.pro`.

Attribute and Uniform Variables for Vertex Shader

This example uses the global wind data (`globalwinds.dat`) that is shipped as part of the IDL distribution. There is also a [2,*n*] vector of (*u*,*v*) pairs, representing the wind velocity at each point.

```
; Get initial positions and wind velocity data.
RESTORE, FILE=FILEPATH('globalwinds.dat', $
   SUBDIRECTORY=['examples', 'data'])

; Set up point grid.
pts = FLTARR(2, 128*64)
FOR i=0, 63 DO BEGIN
   pts[0, i*128:(i+1)*128-1] = x
   pts[1, i*128:(i+1)*128-1] = y[i]
ENDFOR

; Set up per-sample velocity information.
u = REFORM(u, 128*64)
v = REFORM(v, 128*64)
```

```
uv = TRANSPOSE([[u],[v]])
```

This code fragment creates an IDLgrPolygon object with the initial sample (particle) locations and uses `STYLE=0`, which simply draws a dot at each vertex. Instead of placing the velocity data in the shader object, you store it in the graphic object, the polygon, using the SetVertexAttributeData method:

```
; Create graphical object and associate the wind data.
oPoints = OBJ_NEW('IDLgrPolygon', pts, STYLE=0, THICK=3)
oPoints->SetVertexAttributeData, 'uv', uv
```

A time uniform variable is used to determine the amount of displacement of a particle from its original location since time will be multiplied by velocity in the vertex shader program.

```
; Animate and track time.
t0 = SYSTIME(1)
frames = 0L
FOR i=0, 2 DO BEGIN
   FOR time=0.0, 2, 0.01 DO BEGIN
      oShader->SetUniformVariable, 'Time', time
      oWin->Draw
      frames++
   ENDFOR
ENDFOR
```

Warning

Attribute and uniform variable names are case-sensitive, unlike most variable names in IDL.

Hardware Shader Program for Vertex Shader

The vertex program does the majority of the work. The wind velocity data contained in the attribute variable, *uv*, is passed to the vertex program by calling SetVertexAttributeData. The vertex program runs once for every vertex in the primitive. In this case, OpenGL finds the attribute data associated with each vertex as it calls this shader program, and places that attribute data in *uv*. As *uv* is a frequently changing attribute variable, the value will likely be different each time the vertex shader is called.

The uniform variable, *Time*, specifies how long the particle has been moving with the velocity *uv*. Therefore the actual displacement is simply the velocity multiplied by time. The IDL application sets the value of *Time* for each frame in the animation.

```
vertexProgram = [ $
   'attribute vec2 uv;', $
```

```
'uniform float Time;', $
'void main() {', $
   'vec4 vert;', $
   'vert = gl_Vertex + vec4(uv * Time, 0.0, 0.0);', $
   'gl_Position = gl_ModelViewProjectionMatrix * vert;', $
'}']
```

OpenGL sets the special, pre-defined GLSL uniform variable, `gl_Vertex`, to the current vertex position [*x*, *y*, *z*, *w*] for each vertex. The expression `uv*Time` multiplies the 2-element velocity vector by the scalar time value, resulting in another 2-element vector. This result is expanded to a 4-element vector and then added to the vertex location. Finally, the new vertex location is transformed from world to screen space and passed back to OpenGL via the special GLSL variable `gl_Position`.

The fragment program is rather trivial. The only thing this program does is set the color of the point.

```
fragmentProgram = [ $
'void main() {', $
'gl_FragColor = vec4(1.0, 0.44, 0.122, 0.8);', $
'}']
```

Note

Any color set using the COLOR property of the IDLgrPolygon object is ignored since the fragment shader rendering takes precedence over the fixed-function OpenGL pipeline rendering. If the fragment portion of the shader program does not set a fragment color, the fragment (pixel) is drawn with color set to black.

Assign Vertex Shader Program to Shader Object

Since the fragment and vertex shader programs were defined inline, associate them with a newly created shader object using the VERTEX_PROGRAM_STRING and FRAGMENT_PROGRAM_STRING properties. Then assign the shader to the polygon object (*oPoints*).

```
; Set up shader object
oShader = OBJ_NEW('IDLgrShader')
oShader->SetProperty, $
   VERTEX_PROGRAM_STRING=STRJOIN(vertexProgram, STRING(10b)), $
   FRAGMENT_PROGRAM_STRING=STRJOIN(fragmentProgram, STRING(10b))
oPoints->SetProperty, SHADER=oShader
```

The only remaining task is to create the display objects. See `shader_vertexwinds_doc.pro` in the `examples/doc/shaders` directory if needed.

When you run the program, the time loop cycles through the animation three times. The frame rate is printed to the output window when the program finishes. An IDL application could duplicate this example without using shaders by applying the velocity-multiplied-by-time factor repeatedly to all the vertex data and repeatedly updating the vertex data stored in the polygon object. However, this would be a much slower process than the average 200 frames per second achieved by the shader program. The following figure shows a subset of the world map and the final positions of wind vector points.

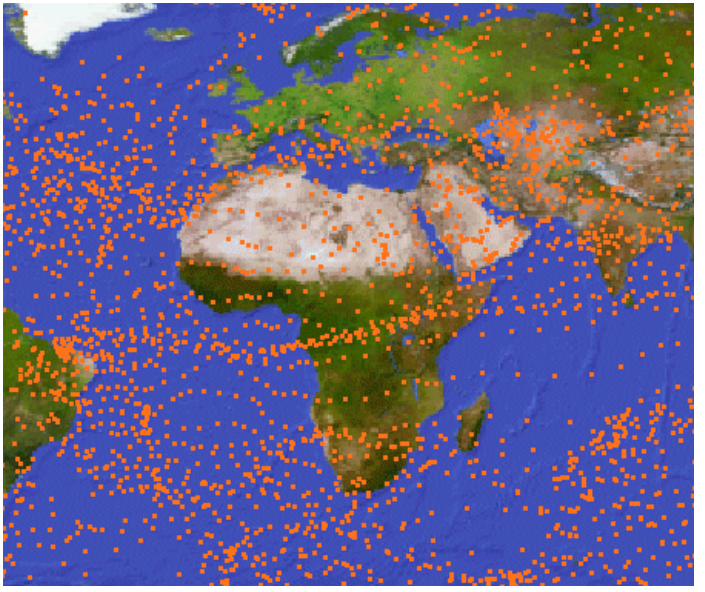


Figure 14-8: Vertex Shader Example Mapping Wind Velocity Data

Lighting Shaders

Shader programs that do not involve computing fragment color based on lighting or shading calculations are typically straightforward and relatively simple. For image filters, the fragment color is determined by the image data, modified by the filter. And drawing simple points requires setting a simple color to tell OpenGL what color to use when drawing the points.

However, when a shader is performing lighting calculations rather than drawing an image, the shader program replaces the fixed, OpenGL lighting calculations. Your shader program code will need to define lighting and shading effects. In general, this is a fairly complex task, but there are tools available to make it a bit easier.

Note

The code in the following example was created with a tool called ShaderGen, by 3Dlabs (http://www.3dlabs.com). It is beyond the scope of this documentation to describe such third party tools. However, an internet search will likely provide several options that allow you to define and adjust lighting parameters and produce usable shader code output.

Beyond defining the characteristics of lights in GLSL code, you need to understand how lights defined in IDL relate to the OpenGL light table.

IDL Lights and the OpenGL Light Table

In IDL there is a limit of 8 active IDLgrLight objects, which you define by their position, direction, color, and other parameters. OpenGL passes these light definitions to the shader program via a pre-defined GLSL array variable called `gl_LightSource`. The shader program then looks up the light definitions in the table and performs the required lighting calculations. The key is determining which IDL light corresponds to a light entry in the `gl_LightSource` table.

The IDLgrLight object LIGHT_INDEX property provides a means of tying a particular IDLgrLight object to an element of the `gl_LightSource` table. When you define a light, you also set the LIGHT_INDEX property to a value between 0 and 7, inclusive, without duplicating a value in any of the lights. You can then pass these indices to the shader program in uniform variables, or simply hard-code it in the shader program.

For example the following code creates a light and tells OpenGL to put the definition for this light in entry 4 of the light source table:

```
oLight = OBJ_NEW('IDLgrLight', TYPE=1, LOCATION=[200,200,500], $
   COLOR=[255,255,255], INTENSITY=0.8, LIGHT_INDEX=4)
```

The shader program then expects to see the definition for this light in entry 4 of the table. Here some shader program code fetches the light characteristics:

```
Ambient  += gl_LightSource[4].ambient * attenuation;
Diffuse  += gl_LightSource[4].diffuse * nDotVP * attenuation;
Specular += gl_LightSource[4].specular * pf * attenuation;
```

Note

In some shader programs, the light needs to be identified by a single integer value (such as `4`) instead of by a light table entry (`gl_LightSource[4]`). In such a case, you can define a uniform variable that contains the light index value and pass this to the shader program. The following example uses this method, defining a uniform variable named *DirectionalLightIndex* with a value of 4 and passing it to the shader program.

Ambient Lights

An ambient light is a bit different from the other lights. IDL does not use a light source to define an ambient light, and an ambient light does not count toward the limit of 8 active lights. Instead, IDL sets the ambient portion of the OpenGL light model state to the desired ambient light color and intensity. (If a light is not defined with a TYPE setting of positional, directional, or spotlight, the light will be considered to be an ambient light by default.) In a shader program, you would add in the ambient light contribution as follows:

```
Ambient = gl_LightModel.ambient;  // Use IDL's ambient light
```

Note

See “IDLgrLight” (*IDL Reference Guide*) for additional information about ambient lights.

Adding Lighting and Shading to a Surface

This example displays an IDLgrSurface, and uses the vertex shader to displace part of it up and down in an animation sequence. It also changes the color of the displaced part slightly for additional emphasis. An ambient light and a positional light illuminate the surface.

Example Code

See `shader_lightsurf_doc.pro`, located in the `examples/doc/shaders` subdirectory of the IDL distribution, for the complete, working example. Run the example procedure by entering `shader_lightsurf_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT shader_lightsurf_doc.pro`.

First create the surface:

```
; Generate surface data and create surface object.
surfdata = BESELJ(SHIFT(dist(100), 50, 50) / 2,0) * 40
oSurface = OBJ_NEW('IDLgrSurface', surfdata, STYLE=2,
COLOR=[200,200,40])

; Create model for the visibile surface and rotate angle
; for good viewing.
oModel = OBJ_NEW('IDLgrModel')
oModel->Add, oSurface
oModel->Translate, -50, -50, 0
oModel->Rotate, [0,0,1], -30
oModel->Rotate, [1,0,0], -60
oModel->Translate, 50, 50, 0
```

Then define the ambient and positional lights. The directional light has an arbitrary light index value (4 in this example) in order to identify it in the shader program.

```
oLightModel = OBJ_NEW('IDLgrModel')
oLightModel->Add, OBJ_NEW('IDLgrLight', TYPE=0, $
  COLOR=[100, 50, 40])
oLightModel->Add, OBJ_NEW('IDLgrLight', TYPE=1,
   LOCATION=[200,200,500], $
   COLOR=[255,255,255], INTENSITY=0.8, LIGHT_INDEX=4)
```

Uniform and Attribute Variables for Lighting Shader

The IDL application (`shader_lightsurf_doc.pro`) creates and passes two uniform variables and an attribute variable containing per-vertex information to the shader program.

- **Displacement** — this attribute variable contains a “displacement mask”, which describes the part of the surface to displace. There is a value for each vertex, where a zero means no displacement will be applied at that point, and a non-zero value describes the magnitude of the relative displacement.

```
disp = FLTARR(100,100)
disp[50:99, 50:99] = MAX(surface)
oSurface->SetVertexAttributeData,'Displacement', $
   REFORM(disp, 100*100)
```

- **DirectionalLightIndex** — this uniform variable identifies the one non-ambient light’s index value that is being passed to the shader program. The value for this uniform variable matches the LIGHT_INDEX value.

```
oShader->SetUniformVariable, 'DirectionalLightIndex', 4
```

Note

The generated shader program requires an integer (`4`) rather than a table entry (`gl_LightSource[4]`) to identify the light. While defining a uniform variable is not a requirement, using *DirectionalLightIndex* in the shader program code makes it easier to understand than hard-coding the number `4`.

- **Time** - this uniform variable is incremented during IDL application execution and the updated value is used within the shader program to vary the amount of displacement with respect to time.

Hardware Shader Program for Lighting Shader

The vertex shader program for this example was largely generated by 3Dlabs’ ShaderGen program. Only a small amount of code needed to be added or modified to make the generated code work with the example IDL application. See the code comments for details.

Example Code

See `lightSurfVert.txt`, located in the `examples/doc/shaders` subdirectory of the IDL distribution, for the complete, working example.

The fragment shader program (`lightsurf.frag`) is very simple:

```
void main() {
gl_FragColor = gl_Color;
}
```

Assign Lighting Shader Program to Shader Object

The vertex shader program is rather long and complex, so it is stored in an external file, as is the fragment shader program. Associate the shader program components with the IDLgrShader object using the VERTEX_PROGRAM_FILE and FRAGMENT_PROGRAM_FILE properties.

```
; Access shader program files.
vertexFile=FILEPATH('lightSurfVert.txt', $
   SUBDIRECTORY=['examples','doc', 'shaders'])
fragmentFile=FILEPATH('lightSurfFrag.txt', $
   SUBDIRECTORY=['examples','doc', 'shaders'])

; Create shader and associate vertex and fragment programs.
oShader = OBJ_NEW('IDLgrShader')
oShader->SetProperty, VERTEX_PROGRAM_FILENAME=vertexFile, $
   FRAGMENT_PROGRAM_FILENAME=fragmentFile

; Associate shader with the surface. You can comment out
; this line to run without the shader program.
oSurface->SetProperty, SHADER=oShader
```

With the appropriate display objects and a FOR loop to increment the uniform variable *Time*, you can visualize the results of applying the shader program lighting calculations to the surface. A detail of the surface during program execution appears in the following figure.

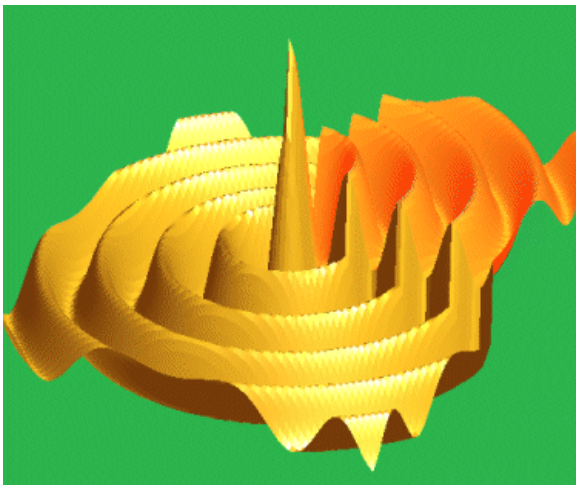


Figure 14-9: Lighting Calculations Applied to Surface Displacement

Multi-texture Shaders

Some applications display multiple 2-D datasets that overlay each other and are layered on an object such as a polygon. When you want to blend the overlying textures in a specific manner, using a shader program provides precise control over how the blending occurs. With a shader multi-texture application, you can specify multiple textures and control how they are displayed relative to each other. Two areas of control are:

1. Texture blending — the shader program controls how the textures are blended with each other and applies simple blending factors that result in an immediate update of the display. The same update in IDL would require re-blending the image and sending the result to the graphics device. This would be required for each modification.
2. Texture coordinate mapping — the application can specify a unique set of texture coordinates for each texture, allowing independent control of the positioning of each texture on the object.

However, if you want to *uniformly* blend images, it may be easier to use traditional IDL methods to create a single image, which can then be used as a texture map. You can combine or "burn" the overlay data into the base image to produce a single image that IDL then displays in the usual, static manner. Suppose your multi-texture example features a map with an overlay of weather data. If the map is an IDL BYTE array with dimensions [3,256, 256] and the cloud data in an IDL BYTE array with dimensions [256, 256], then code to "burn" the clouds into the map might look like:

```
fmap = FLOAT(map) / 255.0
fclouds = FLOAT(clouds) / 255.0
fclouds = TRANSPOSE([ [[fclouds]], [[fclouds]], [[fclouds]] ])
map = BYTE((fmap * (1.0-fclouds) + fclouds) * 255)
```

This code simply increases the amount of white in the image, proportional to the values of the cloud data, and reduces the map color by the same amount. However, to change the blending the IDL application must re-blend the image and send the results to the graphics device each time a blending factor changes. A shader program can handle such multi-texturing tasks with greater flexibility and performance.

Note

Often data for multiple textures will be correctly sized and positioned to map onto a surface in the same way. However, if you need change the position of one texture in relation to others, see "Repositioning Textures" on page 374.

Uniform Variables and Multi-Texture Shaders

When more than one texture is being layered on a surface or polygon, you do not need to use SetUniformVariable to pass the texture data associated with the primary image object to the shader program. (The primary image is the one to which the shader object is attached). The texture map associated with the primary image object data is automatically contained in the reserved uniform variable `_IDL_ImageTexture`. However, you do need to use SetUniformVariable to pass any additional textures to the shader program.

Note

If SetUniformVariable references an IDLgrImage object with dimensions that are not a power of 2, the image will be padded to the next largest power of 2. If the dimensions of the IDLgrImage are larger than MAX_TEXTURE_DIMENSIONS (returned by IDLgrWindow::GetDeviceInfo) then the image will be scaled down to MAX_TEXTURE_DIMENSIONS. Keep this in mind when generating texture coordinates to access the texture map.

Manipulating Multiple Textures Using Shaders

The following multi-texturing shader program example provides the ability to interactively scrape away the section of clouds under the mouse cursor to see the earth below. Because this requires blending only a *section* of the image, using a shader program in this case is far easier than duplicating the outcome using only IDL.

Example Code

See `shader_multitexture_doc.pro`, located in the `examples/doc/shaders` subdirectory of the IDL distribution, for the complete, working example. Run the example procedure by entering `shader_multitexture_doc` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT shader_multitexture_doc.pro`.

Uniform Variables for Multi-texture Shader

This example uses three uniform variables that define the map of the earth, the clouds, and the position of the mouse cursor on the map where you want to reveal the earth below the clouds. These are:

- **Day** — the base image of the map of the earth that is added to the IDLgrModel. This base image object is stored in the reserved uniform variable _IDL_ImageTexture by default and need not be explicitly passed to the shader program using SetUniformVariable.

```
READ_JPEG, 'Day.jpg', day
oDay = OBJ_NEW('IDLgrImage', day)
```

- **Clouds** — this image of the cloud cover is explicitly passed to the shader program using SetUniformVariable in the main IDL application, `shader_multitexture_doc.pro`.

```
READ_JPEG, 'Clouds.jpg', clouds
oClouds = OBJ_NEW('IDLgrImage', clouds)
oShader->SetUniformVariable, 'Clouds', oClouds
```

- **Scrape** — this provides the position of the mouse cursor, which scrapes away a circle of clouds when the scraper has been activated. This information is passed using SetUniformVariable in the OnMouseUp and OnMouseMotion methods of the window observer.

Example Code

The window observer object file is located in `winobserver__define.pro` in the `examples/doc/shaders` subdirectory of the IDL distribution. Run the example

procedure by entering `winobserver__define` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT winobserver__define.pro`.

Hardware Shader Program for Multi-texture Shader

The vertex shader program (`multitextureVert.txt` located in `examples/doc/shaders`) is very simple since the example requires only display-related transformation of the vertices.

```
void main()
{
   gl_TexCoord[0]  = gl_MultiTexCoord0;
   gl_Position     = ftransform();
}
```

This basic vertex program passes along the texture coordinate and then applies a transform to the vertex to correctly position it on the screen. The `gl_TexCoord[0]` is a varying variable that transmits data from the vertex program to the fragment shader program.

Note

If you need to align or change the position of a texture in relation to other textures you can use the SetMultitextureCoord method. See "Repositioning Textures" on page 374 for details.

The fragment shader program (`multitextureFrag.txt` located in `examples/doc/shaders`) uses the three uniform variables to determine which portion of the clouds needs to be removed.

```
uniform sampler2D Clouds;
uniform sampler2D _IDL_ImageTexture;
uniform vec2 Scrape;

void main()
{
vec3 clouds  = vec3(texture2D(Clouds, gl_TexCoord[0].st).r);
vec3 daytime = texture2D(_IDL_ImageTexture,
   gl_TexCoord[0].st).rgb;

   vec3 color = daytime;
   vec2 f = Scrape - gl_TexCoord[0].st;
   f.s *= 2.0;  // aspect ratio correction

   if (length(f) > 0.02)
```

```
        color = mix(daytime, clouds, clouds.r);

    gl_FragColor = vec4(color, 1.0);
}
```

The shader program mixes the map and cloud data according to the cloud intensity, but only when greater than a certain distance away from the specified position (the *Scrape* location). If close enough to the specified position, the program just draws the map color. The shader program is fast enough to let you interactively change the *Scrape* location to reflex the position of the mouse cursor. Attempting the same operation in IDL would likely be too slow to be useful.

Assign Multi-texture Shader Program to Shader Object

You need to supply the program code to the shader object so that it is available to the graphics card when it is needed. To accomplish this, you can use shader object properties VERTEX_PROGRAM_FILE and FRAGMENT_PROGRAM_FILE to associate external shader program components with the shader object.

```
vertexFile=filepath('multitextureVert.txt', $
   SUBDIRECTORY=['examples','doc', 'shaders'])
fragmentFile=filepath('multitextureFrag.txt', $
   SUBDIRECTORY=['examples','doc', 'shaders'])

; Create the shader object, link the shader programs, and
; associate the shader with the base image object, the daytime
; map of the earth (oDay).
oShader = OBJ_NEW('IDLgrShader')
oShader->SetProperty, $
   VERTEX_PROGRAM_FILENAME='multitexture.vert'
oShader->SetProperty,$
   FRAGMENT_PROGRAM_FILENAME='multitexture.frag'
oDay->SetProperty, SHADER=oShader
```

At this point, you can easily add image display code and a window observer to your program and test your multi-texture shader.

When you run `shader_multitexture_doc.pro`, click in the window to turn on the cloud “scraper” and move your mouse cursor to reveal the ground beneath. The

following figure shows the upper Baja peninsula with clouds (left) and without (right) as the shader interactively blends the two textures under the mouse cursor.

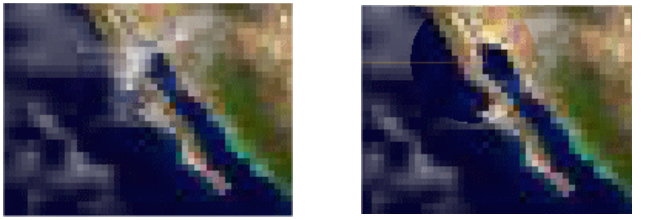


Figure 14-10: Multi-texture Blending Example

Repositioning Textures

When working with multiple textures, the textures may all map the same way onto the object. However, if one texture needs to be repositioned or if you want to animate a texture, you can assign individual texture coordinates to each texture. Using the map and cloud example, you could either shift the position of the clouds or animate the clouds to move across the map.

To achieve such results, you need to supply a different set of texture coordinates for each texture using the IDLgrPolygon::SetMultiTextureCoord or the IDLgrSurface::SetMultiTextureCoord method. This method has the signature of:

```
obj->SetMultiTextureCoord, Unit, TexCoord
```

where *Unit* specifies a texture coordinate unit and *TexCoord* contains the texture coordinates. This effort begins in your IDL application:

```
tcMap =  < code that generates the texture coords >
tcClouds = tcMap
tcClouds[0,*,*] += 0.2  ;; shift the clouds to the west
oPolygon->SetMultiTextureCoord, 0, tcMap
oPolygon->SetMultiTextureCoord, 1, tcClouds
```

The last two lines associate two sets of texture coordinates with the polygon object. Access these texture coordinates in the vertex program where texture coordinate 0 (zero) relates to the map texture and texture coordinate 1 is the cloud texture. Your vertex shader program must collect these and pass them to the fragment shader:

```
gl_TexCoord[0]  = gl_MultiTexCoord0;
gl_TexCoord[1]  = gl_MultiTexCoord1;
```

The fragment shader then uses the appropriate texture coordinate to lookup the color from each texture. That is, it uses `gl_TexCoord[1]` to lookup the cloud texture, and `gl_TexCoord[0]` to lookup the map texture.

```
vec3 clouds  = vec3(texture2D(Clouds,
   gl_TexCoord[1].st).r);
vec3 map = texture2D(Map, gl_TexCoord[0].st).rgb;

vec3 color = mix(map, clouds, clouds.r);
gl_FragColor = vec4(color, 1.0);
```

Thus, you can use two sets of texture coordinates to control the display of two different textures.

Rotating Earth with Multiple Textures

This example loads three images, a base day image of the earth, a night image and an image of clouds, into textures. It then draws the rotating earth showing a day scene on one side and night scene (lights of big cities) on the other.

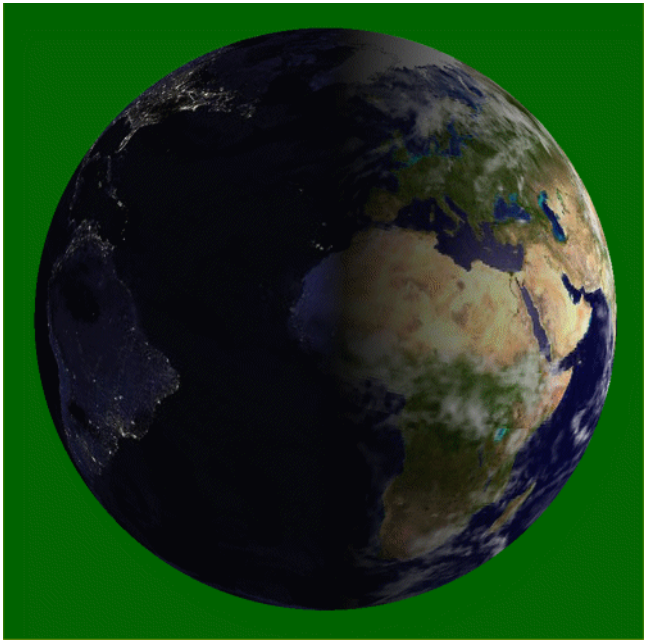


Figure 14-11: Sample Image from Multi-texture Shader Application

In the IDL code:

- Create three IDLgrImage objects to hold the daytime, nighttime and cloud textures. Assign the object references for these image objects to uniform variables, using the IDLgrShader::SetUniformVariable method.

```
; Tell the shader program about our textures.
oShader->SetUniformVariable, 'EarthDay', oDay
oShader->SetUniformVariable, 'EarthNight', oNight
oShader->SetUniformVariable, 'EarthCloudGloss', oClouds
```

- Use the SetMultiTextureCoord method for IDLgrPolygon to set texture coordinates (*tc*) for the textures.

```
oEarth->SetMultiTextureCoord, 0, tc
```

Note

If the textures did not share the same coordinates, you could call SetMultiTextureCoord multiple times. See "Repositioning Textures" on page 374 for additional information.

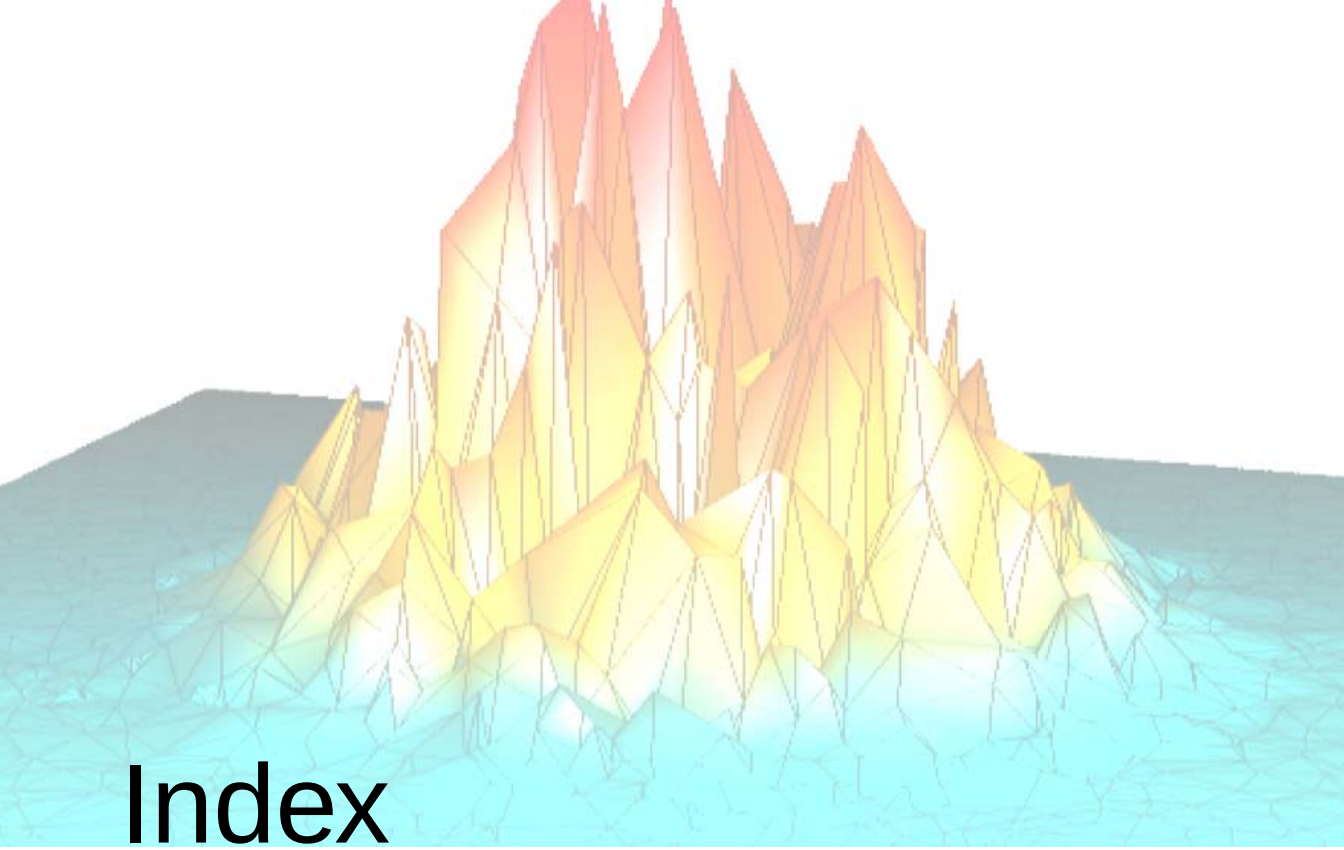
In the Shader Program:

- In the vertex program (`earthVert.txt`), fetch the texture coordinates for both the daytime and nighttime textures from the predefined GLSL uniform variable `gl_MultiTexCoord[n]`, where *n* corresponds to the numbers used in the IDLgrPolygon::SetMultiTextureCoord method calls. These texture coordinates are passed to the fragment shader with varying variables.
- The fragment shader (`earthFrag.txt`) then decides on what side of the earth the fragment is on, and chooses the appropriate texture and texture coordinates to use to look up the texel value to use as the fragment color.

This shader program was taken directly from Chapter 10 of the "Orange Book" ("OpenGL Shading Language", Second Edition, by Randi J. Rost) and required no modifications to work with the IDL application, `shader_earthmulti.pro`.

Example Code

See `shader_earthmulti.pro`, located in the `examples/doc/shaders` subdirectory of the IDL distribution, for the complete, working example. Run the example procedure by entering `shader_earthmulti` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT shader_earthmulti.pro`. The associated shader program files `earthVert.txt` and `earthFrag.txt` are located in the same directory.



Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

T

U

V

W

X

Z