SC38-0496-06

IBM Visualization Data Explorer

User’s Guide

Version 3 Release 1 Modification 4

IBM Visualization Data Explorer

SC38-0496-06

User's Guide

Version 3 Release 1 Modification 4

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xiii.

Seventh Edition (May 1997)

This edition applies to IBM Visualization Data Explorer Version 3.1.4, to IBM Visualization Data Explorer SMP Version 3.1.4, and to all subsequent releases and modifications thereof until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for readers’ comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation
Thomas J. Watson Research Center/Hawthorne
Data Explorer Development
P.O. Box 704
Yorktown Heights, NY 10598-0704
USA

If you send information to IBM, you grant IBM a nonexclusive right to use or distribute that information, in any way it believes appropriate, without incurring any obligation to you.

Contents

Figures

Tables

Notices

Products, Programs, and Services

Trademarks and Service Marks

AViiON	Data General Corporation
DEC	Digital Equipment Corporation
DGC	Data General Corporation
Graphics Interchange Format (GIF)	CompuServe, Inc.
Hewlett-Packard	Hewlett-Packard Company
HP	Hewlett-Packard Company
iFOR/LS	Apollo Computer, Inc.
Motif	Open Software Foundation
NetLS	Apollo Computer, Inc.
Network Licensing Software	Apollo Computer, Inc.
OpenWindows	Sun Microsystems, Inc.
OSF	Open Software Foundation, Inc.
PostScript	Adobe Systems, Inc.
X Window System	Massachusetts Institute of Technology

Copyright notices

About This Guide

This manual is a guide to using IBM Visualization Data Explorer* for:

- manipulating and controlling data visualizations
- importing various kinds of data for visualization
- creating and customizing visual programs with the Visual Program Editor
- using the Data Explorer scripting language to create visual programs.

Who Should Use It

This Guide is intended for users of different degrees of knowledge and experience with graphical programs:

Non-programmers	The non-programmer can learn how to use previously created visual programs to examine data sets (e.g., modifying one or more inputs to a visual program and saving and restoring the results).
Programmers	The programmer can learn how to use: • the visual programming interface to create visual programs and applications. • the scripting language to create visualizations.

This Guide assumes that you have some knowledge of the operating system and the X Window System** being used, as well as of OSF**/Motif**. For more information, see *IBM AIXwindows User's Guide* or the appropriate window system documentation.

In this Guide, any reference to the X Window System means any window server that supports the X11 protocol, including Sun's OpenWindows**.
The Motif window manager, mwm, has been used in many figures and examples in this Guide. Please use the appropriate window manager for your system, such as vuewm (Hewlett-Packard), 4dwm (SGI), or olwm (Sun). Since title bars and window borders are features of a window manager, the appearance of your windows may differ slightly from those in the figures and examples.

How To Use It

- Chapter 1, "Overview" on page 1, describes IBM Visualization Data Explorer—an integrated visualization environment—and its main features.
- Chapter 2, "Introduction to Visualization" on page 7 introduces the basic terminology and working principle of Data Explorer.
- Chapter 3, "Understanding the Data Model" on page 15, presents a formal description of Data Explorer's underlying data model. (Users who do not require such a description, however, should find the informal treatment in Chapter 2, "Introduction to Visualization" on page 7 sufficient for their purposes.)
- Chapter 4, "Data Explorer Execution Model" on page 37 describes the Data Explorer execution model.
- The next five chapters deal with various aspects of the Data Explorer graphical user interface:
  – Chapter 5, "Graphical User Interface: Basics" on page 57
  – Chapter 6, "Graphical User Interface: Important Windows" on page 73
  – Chapter 7, "Graphical User Interface: Control Panels, Interactors, and Macros" on page 127

  - Chapter 8, “Graphical User Interface: Menus, Options, and the Message Window” on page 155
  - Chapter 9, “Graphical User Interface: For Advanced Users” on page 177

  If you intend to use only existing visual programs, see
  - 5.4, “Executing a Visual Program” on page 67
  - 6.1, “Using the Image Window” on page 74
  - 6.3, “Using the Colormap Editor” on page 119.
  - 7.1, “Using Control Panels and Interactors” on page 128
- Chapter 10, “Data Explorer Scripting Language” on page 187 presents a more traditional approach to creating data visualizations. In this connection, see also Chapter 1, “Data Explorer Tools” on page 1 in *IBM Visualization Data Explorer User’s Reference*.
- Appendix A, “Using Data Explorer: Some Useful Hints” on page 211 describes some of the ways to use Data Explorer more effectively.
- Appendix B, “Importing Data: File Formats” on page 241 discusses various data formats that Data Explorer can import, including Data Explorer’s native format.
- The remaining appendixes contain information of varying interest to different users:
  - Appendix C, “Environment Variables and Command Line Options” on page 291
  - Appendix D, “User Interface Configuration” on page 299
  - Appendix E, “Data Explorer Fonts” on page 307
  - Appendix F, “Data Explorer Colors” on page 313
  - Appendix G, “Accelerator Keys” on page 315.

Typographic Conventions

Boldface	Identifies commands, keywords, files, directories, messages from the system, and other items whose names are defined by the system.
Italic	Identifies parameters with names or values to be supplied by the user.
`Monospace`	Identifies examples of specific data values and text similar to what you might see displayed or might type at a keyboard or that you might write in a program.

Related Publications and Sources

IBM Publications

- *IBM Visualization Data Explorer User’s Guide*, SC38-0496

  Details the main features of Data Explorer, including the data model, data import, the user interface, the Image window, and the visual program editor. and the scripting language. Of particular interest to programmers: chapters on the data model and the scripting language.

- *IBM Visualization Data Explorer User’s Reference*, SC38-0486

  Contains detailed descriptions of Data Explorer’s tools.

  Note: Consult this reference if you are creating visual programs or scripts.

- *IBM Visualization Data Explorer Programmer’s Reference*, SC38-0497

Contains detailed descriptions of the Data Explorer library routines.

Note: Consult this reference if you are writing your own modules for Data Explorer.

Non-IBM Publications

The following treat various aspects of computer graphics and visualization:

Adobe Systems Incorporated, *PostScript Language Reference Manual*, 2nd Ed., Addison-Wesley Publishing Company, Massachusetts, 1990.

Aldus Corporation and Microsoft Corporation, *Tag Image File Format Specification, Revision 5.0*, Aldus Corporation, Washington, 1988.

Arvo, Jim, ed., *Graphics Gems II*, Academic Press, Inc., Boston, Massachusetts, 1991.

Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F., *Computer Graphics: Principles and Practice*, Addison-Wesley Publishing Company; Massachusetts, 1990.

Friedhoff, Richard M., and Benzon, William, *Visualization: The Second Computer Revolution*, New York, Harry N. Abrams, Inc., 1989.

Glassner, Andrew, ed., *Graphics Gems*, Academic Press, Inc., Boston, Massachusetts, 1990.

Hill, F.S., Jr., *Computer Graphics*. Macmillan Publishing Company, New York, 1990.

Kirk, David, ed., *Graphics Gems III*, Academic Press, Inc., Boston, Massachusetts, 1992.

Robin, Harry, *The Scientific Image: from cave to computer*, Harry N. Abrams, Inc., New York, 1992.

Rogers, David F., *Procedural Elements for Computer Graphics*, McGraw-Hill Book Company, New York, 1985.

Rogers, David F. and Adams, J.Alan, *Mathematical Elements for Computer Graphics*, 2nd Ed., New York, McGraw-Hill Book Company, 1990.

SIGGRAPH Conference Proceedings, Association for Computing Machinery, Inc.: A Publication of ACM SIGGRAPH, New York, various years.

Tufte, Edward, *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, Connecticut, 1983.

Other sources of information

For additional ideas, consult the "DX Repository," available through anonymous FTP (`ftp.tc.cornell.edu.` in directory `pub/Data.Explorer`), and gopher (`ftp.tc.cornell.edu.` port 70). This public software resource includes information and visual programs contributed by Data Explorer users from around the world. We encourage you to contribute your innovations and ideas to the Repository, in the form of new modules, macros, visual programs, and tips and tricks you discover as you learn and master Data Explorer.

On the Internet, the newsgroup `comp.graphics.apps.data-explorer` is used by customers around the word to share information and ask questions. This newsgroup is also followed by Data Explorer developers.

If you have access to the World Wide Web, you can find the Data Explorer home page at `http://www.almaden.ibm.com/dx/`.

New Features in Data Explorer Version 3.1.4

User Interface

New Startup Behavior

With this release, when you type `dx`, a different initial panel will appear, giving you access to various parts of Data Explorer, such as the Data Prompter, the Tutorial, the Visual Program Editor, etc.

To bypass the Startup window and go directly to the Visual Program Editor (as in previous versions of Data Explorer) either type `dx -edit` at the prompt or set your `DXARGS` environment variable to `-edit`.

New Save Image Dialog in Image Window

The Save Image dialog has been improved to make it easier for users to save images at a specific size on the printed page.

New Data Prompter

The Data Prompter has a new initial window, which allows you to specify what kind of data you have (dx format, image format, ...). You can access the Data Browser from this window to view your data before attempting to import it. If you have “general array” format data, then choose the “Grid or Scattered file” button, which will lead to the interface which was called the Data Prompter in previous versions of Data Explorer.

Once your data has been imported, there is an option to either describe or visualize your data. If you choose the “Test Import” option, then the data will be imported and characteristics about it (such as dimensionality, number of points, ...) will be reported to you. If you choose the “Visualize Data” option, then a general purpose visual program will be run on your data. You can then inspect and modify this visual program.

Pages

In the VPE, you can now segment your visual program into “pages”, which are disconnected sets of modules. Modules in one page communicate with modules in other pages using transmitters and receivers. You can name pages and control the ordering of pages. See the Edit menu of the VPE.

Annotation

You can add comments directly onto the canvas of the VPE. This option is available from the Edit menu of the VPE.

Optimizing Caching

There is now an option in the Edit menu of the VPE called Output Cacheability -> Optimize Cache. If you choose this option, Data Explorer will use a heuristic to optimally set the caching behavior of each tool in the visual program.

Changes to Get and Set modules

In this release, the Get and Set modules have been replaced by GetLocal/SetLocal and GetGlobal/SetGlobal. Briefly, the difference between these pairs of modules is that the Global pair maintains state between executions, while the Local pair do not. (Remember that a single loop in the visual program is considered a single execution). If you do not do anything to modify your visual program, any Get/Set pairs which you have will be replaced by GetGlobal and SetGlobal, and the visual program will run as it did before. However, in many cases you can replace your Get/Set pair by GetLocal and SetLocal for performance advantages. One way to know if you can do this is if you are using the First module to reset Get on each execution. If so, then you can certainly replace your Get and Set by GetLocal and SetLocal (and the First module is no longer necessary, as GetLocal automatically resets on each new execution).

See “GetLocal” on page 151, “SetLocal” on page 300, “GetGlobal” on page 149, and “SetGlobal” on page 299 in *IBM Visualization Data Explorer User’s Reference* for more detailed information.

Assign Get/Set Scope → Convert All modules under the Edit pulldown in the VPE is available for helping change Get/Set modules to the new GetGlobal/SetGlobal, GetLocal/SetLocal options.

New Window Management Functionality

The SuperviseWindow and SuperviseState modules (see “SuperviseState” on page 332 and “SuperviseWindow” on page 336 in *IBM Visualization Data Explorer User’s Reference*) implement important new functionality for users, allowing you much more control over the effect of mouse and keyboard actions in a Display window. This allows you not only to define the behavior for given mouse or keyboard events, but also allows you to implement direct interaction without the use of the Image tool. Thus direct interaction is now possible without the Data Explorer User Interface.

Hardware Rendering

True transparency is now supported for OpenGL platforms (previously only screen-door transparency was supported).

Anti-aliasing of lines, and multiple-pixel-width lines is now supported in OpenGL and GL. To specify anti-aliasing of lines, pass the object to be rendered through the Options module, setting an attribute of “antialias” with a value of “lines”. To specify multiple pixel width lines, pass the object to be rendered through Options, setting an attribute of “line width”, with value set to the number of pixels.

DXLink

A number of new routines which allow execution of named macros, and control over window management (e.g. opening and closing image windows) have been added. See Chapter 16, “DXLink Developer's Toolkit” on page 157 in *IBM Visualization Data Explorer Programmer’s Reference*.

Changed Modules

This section contains only summaries. See *IBM Visualization Data Explorer User's Reference* for details.

AutoAxes

There are new inputs to AutoAxes which allow you to explicitly specify tick locations, and optionally, to specify labels to be associated with those locations. Thus for example, if one of your axes represents the months, with integers 1 to 12, you could indicate to AutoAxes to place ticks at each integer from 1 to 12 and to label them "jan", "feb", ... Plot and ColorBar similarly now allow you to specify tick location and label.

AutoAxes no longer scales the input object. This should make using probes in an image which includes axes easier. There is a break with backward compatibility in that for some viewpoints the axes may not lie completely within the image; you will need to change the viewpoint to be slightly farther away from the object.

AutoGlyph, Glyph

"cube" and "square" glyphs have been added.

ColorBar

Has new inputs which allow you to specify precise tick locations and labels for the ticks (see AutoAxes).

Compute, Compute2

Now perform string operations, such as strcmp, strlen, strstr, etc. Compute also provides a "random" function.

Display

Now has output called "where". This is the identifier for the window into which the image was displayed. See ReadImageWindow on page xxxii.

Export

Now exports VRML 2.0 data.

Get/Set

Are replaced by GetLocal/SetLocal and GetGlobal/SetGlobal. See "Changes to Get and Set modules" on page xxix.

Histogram

Now will create 2- and 3-dimensional histograms for vector data.

Image

Now has new output called "where". This is the identifier for the window into which the image was displayed. See ReadImageWindow on page xxxii. Also, if you choose the Rerender Image option to render the image at a higher (or lower) resolution, screen objects such as captions and color bars will now be WYSIWYG. See ScaleScreen on page xxxii.

Include

Has a new input called "pointwise". If this input is set, then Include will remove the connections of the input before removing points. Thus only those positions with data values outside the specified range will be removed. The default behavior (and previous behavior) is to also remove all connections (and positions referenced by those connections) containing at least one invalid point.

Inquire

Has several new inquiries: “is image”, “is connection”, “object tag”, “connection type”, “valid count”, “invalid count”

Integer, IntegerList, Scalar, ScalarList, Vector, VectorList

Each of these data-driven interactors now has a “refresh” input. This input resets the interactor, as if it is running for the first time, regardless of the current output value with respect to the range of the current input to the interactor.

Pick

Has new inputs **`locations`** and **`camera`**. These inputs are intended for use when picking is done in an image window created using SuperviseWindow, rather than the Image tool.

Plot

Has new inputs which allow you to specify precise tick locations and labels for the ticks (see AutoAxes on page xxxiii).

ReadImage

ReadImage supports miff images. If the .miff file contains a sequence of images, they will be read in as an image series. ReadImage has a new input, “colortype”, which allows control over the pixel format used internally to represent the loaded image. It also has a new input, **`delayed`**, which specifies whether images stored in image-with-colormap format should be imported as a delayed colors image.

WriteImage

Supports MIFF output, which is a run-length-encoded format that supports image sequences. Supports GIF output of any image.

New Modules

This section contains only summaries. See *IBM Visualization Data Explorer User’s Reference* for details.

AutoGrid

This module provides much of the function of the existing module Regrid. However it automatically will construct a grid for you rather than requiring you to create one.

Categorize

Categorizes data, replacing the categorized component by a list of indices into a lookup component. Among other things, allows string data to be categorized.

CategoryStatistics

Computes statistics on categorical data.

ChangeGroupMember

Allows you to insert a new member into a group, or replace a member of a group.

ChangeGroupType

Allows you to change the type of a group (for example from a generic group to a series)

CopyContainer
Allows you to copy the header of an object.

Describe
This module is used to "describe" an input object. For example, it will tell you the structure of the object (how many data points, the bounding box, etc.) It can also tell you whether or not it is ready to be rendered (i.e., is a valid input to the image tool).

DXLInputNamed
Similar to DXLInput, but allows you to specify the name of the DXLink variable by passing it into the module rather than via the configuration dialog box for the module.

ImportSpreadsheet
Imports tabular (spreadsheet) data.

Legend
Creates a legend bar, which is similar to a color bar, but which associates colors with strings.

The Legend module also accepts a colormap for the second parameter (colorlist). If a colormap is given, then the colors corresponding to the integers *0, ..., n-1*, where *n* is the number of items in the stringlist (first parameter) are assumed.

Lookup
Use one object to lookup the value of another object in a field. This module is especially useful with categorical data.

QuantizeImage
Allows you to create a "delayed colors" image from any image. You specify the number of colors to use and the module will choose the best set of colors to represent the image.

ReadImageWindow
Allows you to obtain the image (that is, the field of pixels) from a Display or Image window. (ReadImageWindow is called internally by the Image tool, when necessary to save a displayed image)

ScaleScreen
Scales all screen objects (i.e. captions, color bars, text glyphs) by a specified amount. Used internally by the Image tool to make sure that rerendered images remain WYSIWYG.

SimplifySurface
Reduces the number of triangles in a surface.

SuperviseState
Used with SuperviseWindow to create and manage windows. This pair of modules allows you to directly specify what actions should take place for a given mouse or keyboard event in the window. This is in contrast to the use of the Image window, where mouse actions are predefined by Data Explorer (i.e. rotation or pan/zoom mode).

SuperviseWindow
See SuperviseState on page xxxii.

Backward Incompatibilities

There are a few backward incompatibilities with previous versions of Data Explorer.

AutoAxes
No longer scales the object to be smaller. This means that you may need to zoom out a bit in order to see all of the axes labels.

DXLink
Because of the new startup behavior (see “New Startup Behavior” on page xxviii), DXLink programs may need to add “-edit” or “-image” to the startup command string.

Get and Set
Have been replaced by GetGlobal, GetLocal, SetGlobal, and SetLocal. See “Changes to Get and Set modules” on page xxix.

Plot
There is a backward incompatibility with regard to the aspect parameter of Plot. This parameter now defaults to 1.0, meaning that the ratio of the y to x axis length will be made equal to one. Previously, no scaling was automatically done if aspect was not set. The previous default behavior can be obtained by specifying aspect as “inherent”.

ReadImage
The ReadImage module now will store all images read in as three bytes, rather than than three floating point numbers. This will only affect visual programs in which the colors themselves are expected to be floats. This default behavior can be overridden either with an environment variable or an input parameter. In addition, images which are stored in a format such that the colors are specified as lookups into a table will be read in as “delayed colors”. (Some modules may not perform properly on delayed colors images.) This default behavior may also be overridden with either an environment variable or an input parameter. See “ReadImage” on page 250 in *IBM Visualization Data Explorer User’s Reference* for more information.

Transmitters and Receivers
Version 3.1.4 prevents Transmitter and Output nodes, and Receiver and Input nodes from sharing names. These name collisions were permitted in earlier versions and could lead to incorrect behavior. Now, colliding nodes will be renamed automatically and you will be notified.

Save Image from Image Window
By default, SaveImage in PostScript format will now nearly fill the page, and will automatically choose portrait or landscape orientation.

WriteImage (and Image)
A gamma correction factor of 2 is applied to all images when they are written out. This can be changed if desired by using the format parameter of WriteImage or the SaveImage dialog of the Image tool. Previously, images were not gamma corrected when saved.

DXSHMEM environment variable
In versions of Data Explorer prior to 3.1.4, DXSHMEM, if set to anything, would force shared memory to be used. In version 3.1.4, DXSHMEM must be set to anything other than -1 for shared memory to be used; if

set to -1, then the data segment will be extended, for architectures for which this is permissible.

HTML Documentation

For the HTML version of the complete Data Explorer documentation, point your web browser at `$DXROOT/html/index.htm`.

Fixes

Creating an outboard module on an SGI will no longer fail due to lack of resources resulting from fork.

SelectorList interactors can now contain more entries.

Cache management has been improved.

It is now possible to save texture-mapped images, both from the Image window and by capturing the output of Display. See ReadImageWindow on page xxxii.

Chapter 1. Overview

This book describes the IBM Visualization Data Explorer*, which you can use in a workstation environment. Data Explorer is a visualization system that can be used in many application areas and with a variety of data representations to extract useful information from complex data.

1.1 Overview of Data Explorer

The Data Explorer graphical user interface allows end users to perform tasks at various levels of sophistication. For example, a user can use the user interface to apply data and adjust input values to an existing visualization process. A slightly more advanced user can construct a new visualization process, called a visual program, by connecting a network of Data Explorer's modules. An expert programmer can create new modules, using C or FORTRAN, for use with the system modules. Besides the user interface, Data Explorer also provides a scripting language interface, for users who want to build their own visualization functions in a more traditional programming style.

Data Explorer's graphical user interface provides an integrated online help facility. This facility provides users with online access to the Data Explorer user manuals, as well as with context-sensitive help information. In addition to the help information provided with Data Explorer, the online help facility allows users to document various aspects of their particular visual programs. Other users of these visual programs then have online access to this program-specific documentation.

Data Explorer provides an extensive set of modules that you can use to visualize your data. For example, the Isosurface, Streamline, and AutoColor modules perform the standard visualization functions of creating constant-value surfaces, tracing particle paths through velocity fields, and coloring objects based on a data value, respectively.

In addition to these expected functions, Data Explorer also provides tools to perform more sophisticated manipulation of data. The Map module is a general purpose module that can map a data field onto an arbitrary object—whether it is a streamline, an isosurface, or even another data field's computational mesh. The Compute module can perform arithmetic or trigonometric operations point-by-point not only on your data but also on the grid itself. Thus warping a grid, for example, is a simple matter of entering an expression.

Even standard tools, such as Isosurface, operate on multiple types of input grids. For example, if the input field to Isosurface is 2-dimensional, the module automatically creates contour lines.

The Data Explorer renderer can handle opaque or translucent surfaces, translucent volumes, and opaque or translucent lines or points—all in the same image. In addition, data on different computational or observational grids can be visualized together, allowing you to correlate disparate data fields without requiring you to force the data onto the same grid.

The power and interoperability of the modules is possible because of the underlying data model, which is capable of describing a wide variety of types of input data. Because the data itself is self-describing, modules can be flexible in the types of data they accept, and can perform their actions appropriately based on their input.

1.2 System Structure

Data Explorer is designed as a client-server model. The Data Explorer client-server architecture incorporates system components such as TCP/IP, sockets, X Window System, and Motif.

In this client-server model, the user interface is the client. The executive, modules, and data management components, often referred to collectively as the executive, make up the server portion. The user interface client can be on a different platform from the server (executive), and the executive can run on multiple platforms simultaneously (distributed processing). Data Explorer allows you to switch among servers running on different hardware platforms.

The Data Explorer system can be thought of as consisting of four "layers," each with its own defined interface. These layers are described in the order in which you are likely to encounter them:

- Graphical user interface
- Executive
- Modules
- Data management.

Graphical User Interface

The graphical user interface is built upon the X Window and Motif standards. These tools manage multiple application windows that allow a user to create and control the visualization process easily and effectively. The graphical user interface provides two levels of service. First, non-programmers or users with fixed requirements can execute previously created visual programs. These visual programs may consist of various menus, dials, sliders, and other interactors that provide fixed functions. Second, programmers can create customized visualizations by using the interface to interconnect modules in flexible ways, and to create new combinations of modules in the form of macros.

The Data Explorer graphical user interface lets you create or work with a visual program to easily realize sample, select, and transform data during visualization. You can use the Visual Program Editor (VPE) to create new scenarios by simply connecting module icons on the screen in any logical sequence.

Data Explorer provides the following primary windows:

Visual Program Editor	Lets you create and alter visual programs.
Control Panel	Lets you set and control the variable input parameters of the tools used in a visual program.
Image Window	Displays the image created by a visual program and allows direct interaction with the visualized image.
Help Window	Provides online access to the Data Explorer user manual and context-sensitive help information.

Data Explorer provides a Colormap Editor window that lets you map colors to specified data values and display the results in the visual image. The system also provides a Sequencer window, which has many uses, including controlling how a sequence of images is displayed (with forward and backward direction, repetition, and so on).

These windows are discussed in detail in Chapter 5, “Graphical User Interface: Basics” on page 57. In addition, Data Explorer provides two stand-alone utilities:

`Data Prompter:` a point-and-click interface for describing a data set for importing. (See *IBM Visualization Data Explorer QuickStart Guide*.)

`Module Builder:` a point-and-click interface for describing the interface to a user-written module. The Module Builder creates the necessary makefiles and a template **`.c`** file for the module. (See *IBM Visualization Data Explorer Programmer’s Reference*.)

Executive

The executive is the component of the system that manages the execution of the modules specified in the scripting language. This scripting language is generated by the graphical user interface to invoke visualization functions for visual programs. Users can also use the scripting language to write their own programs, as described in Chapter 10, “Data Explorer Scripting Language” on page 187.

Modules

Data Explorer provides an extensive, powerful set of highly interoperable visualization modules. The modules used for visualization functions are available:

- As nodes, through the use of their icons in a visual programming network.
- As function calls, available in the scripting language interface provided by the executive layer.
- For integrated applications, as part of the visualization library programming interface. (See 12.10, “Module Access” on page 127 in *IBM Visualization Data Explorer Programmer’s Reference* for information on this use of modules.)

Data Management

The data management layer is the portion of the programming interface that provides modules with access to the data model, which is discussed in Chapter 3, “Understanding the Data Model” on page 15. This layer includes general system services as well as routines for creating and managing the set of data objects. The data management layer also provides an application programming interface (API) for adding new modules to Data Explorer and for accessing the power and flexibility of the data model.

Detailed information on this API can be found in *IBM Visualization Data Explorer Programmer’s Reference*.

How the Data Model Facilitates Interoperability

The Data Explorer data model is not simply a convenient way to represent data objects. It also allows Data Explorer tools to be more powerful than they would be otherwise.

Tools can be used in multiple ways, because the components of the data set are described using a common structure. There is no distinction between “data,” “positions,” and “colors” in how they are represented within a Data Explorer field object. (For more information on Data Explorer fields, see Chapter 3, “Understanding the Data Model” on page 15.) For example, you can use the Compute module to operate on the data to extract the magnitude, or x component of a vector (e.g., operate on the positions of a grid to warp the grid or on the colors of a field to negate an image). This also means that the user has the capability of modifying or inspecting all aspects of a data object.

Tools can be used on any object in Data Explorer; there is no distinction between "data objects" and "geometry objects." An isosurface or an image is represented in the same way in which an imported data field is represented. So for example, you can:

- create an isosurface (contour lines) of a mapped isosurface
- create an isosurface from an image
- map onto glyphs, streamlines, isosurfaces, etc.

The Data Model also ensures that the fidelity of the original data is maintained throughout the visualization process. In particular, all of the following are preserved throughout:

- the original coordinate space of the data
- the original range of data values (not scaled to 0 to 255, for example)
- attributes of the data (dependency on positions or connections)
- the presence of missing or invalid data.

Finally, the data model ensures that Data Explorer users and developers can add new components or new attributes without modifying current modules.

Chapter 2. Introduction to Visualization

This chapter is not a substitute for the detailed information in the rest of this Guide, but it does summarize some important terms and concepts that may be new to you if you have not used a scientific visualization application before. So we suggest the following:

- Read this section first, concentrating on topics that are unfamiliar.
- Follow the tutorials in *IBM Visualization Data Explorer QuickStart Guide*.
- Start using Data Explorer. A good place to begin is the set of example “networks” (or “visual programs”) in the directory **`/usr/lpp/dx/samples/programs`**. You can open up any visual program file and study how the different modules are interconnected and then run the visual programs to observe the visual output.
- Use the online Help system to get more information about these example visual programs and Data Explorer tools. This system also contains hypertext references to additional information.

  The printed documentation contains detailed information, including graphics, sample code, and data examples.

2.1 Terminology

Many of the terms used in Data Explorer are borrowed from traditional scientific disciplines, others come from computer graphics, and a few have been coined by the Data Explorer software developers for lack of any widely accepted term. Important Data Explorer terms are defined in the Glossary.

Rendering

The process of *rendering* an image involves a computer calculation of the amount of light falling on each visible surface of the objects in the “scene,” as seen from the point of view of the computer “camera” (the viewer's eye point). During the rendering process, surface properties of objects are taken into account as are the colors of both the objects and the “lights” shining on them. In other words, a computer graphics renderer samples the scene in front of the camera at the resolution of the computer monitor on which the scene is to be displayed. Its sample space is the 3-dimensional “world” containing the objects. But the image renderer does not create a 3-dimensional picture; it only calculates the colors of the dots that can be seen on the 2-dimensional monitor screen from the chosen point of view. Any parts of objects that cannot be seen from that point of view are neither sampled nor rendered, nor are they stored in the image file or displayed on the monitor. This 2-dimensional image may appear 3-dimensional to our eyes because of shading, occlusion of distant objects by closer ones, and other visual cues that, in the real world, indicate dimensionality. Like any image, it is a representation, however real it may appear.

Positions and Connections Dependence

The concept of sampling should be familiar to anyone who has ever collected data on some kind of grid. For example, a botanist may lay down a series of square grid markers over an area of interest then count the numbers of species of grasses growing inside each grid square. The number so collected becomes a sample value or datum associated with that grid marker. A single number like this, whether floating point or integer, is called a *scalar*. If the wind velocity and direction at, say, the center of each grid square is also measured, the botanist would record a *vector* quantity as a second datum sampled at the same place. A vector encodes both direction and magnitude with two or more numeric “vector components.”

In this example, the locations of the corners of each grid marker are recorded as an array of 2-dimensional coordinates that define the sampling area dimensions and the sampling resolution. In computer graphics terms, these spatial location points are called *vertices* (singular: vertex); in Data Explorer, they are referred to as “positions”. Loosely, everyone calls them “points.”

Four coordinate positions can be connected by a quadrilateral to define a grid *element*. The quadrilateral itself is called a *connection* in Data Explorer (we will discuss other connection types in a moment). Since the botanist collected one set of data per grid element, such data are termed *connection-dependent data*. This implies that the data value is assumed by Data Explorer to be constant within that element.

Consider another technique for data sampling: on a larger scale, remote-sensing satellites can resolve various features of the Earth down to some finite level of resolution. In this case, the grid positions are identified by a latitude-longitude coordinate pair, and the data values may encode such things as surface reflectance in the ultraviolet. By associating each data value with a latitude-longitude position, we produce *position-dependent data*.

This implies that data values should be interpolated between positions, using the connections (grid) if one is present. Data Explorer works equally well with position-dependent and connection-dependent data (see Figure 1 on page 10). Generally, the decision about which dependency the data has is made by you at the time of data collection or simulation. (There is a simple way in Data Explorer to convert either dependency to the other. See “Post” on page 242 in *IBM Visualization Data Explorer User’s Reference*.)

We can extend our data sampling into three dimensions where appropriate. In that case, we identify each grid position with three coordinates. These coordinates form the corners of “volumetric” elements and the entire sample space is called a *volume*. A volumetric element may be a rectangular prism (like a *cube*) or a *tetrahedron* (a solid with four triangular faces, not necessarily equilateral).

Connections and Interpolation

In the cases just discussed, we made the implicit assumption that there is a logical connectivity between adjacent members of our 2-dimensional or 3-dimensional grid positions. The path connecting grid positions is called a *connection* in Data Explorer. For a surface (2- or 3-dimensional positions connected by 2-dimensional connections), we could choose to make triangular or quadrilateral connections (i.e., *triangles* or *quads*). Quads require four positions for each connection and triangles three. Data Explorer supports these *element types* as well as cubes, tetrahedra, and lines.

Suppose we first choose to link adjacent positions in the botanist’s sample area with *line* connections. The grid markers were 1 meter on a side. Given a sampling area of 5 meters by 3 meters, the entire sample would be 15 meters square; there would be 24 positions (6 in X, and 4 in Y). On such a plot, we see that a position located at [x=0,y=0] is connected to its neighbor at [x=1,y=0]. We can imagine that it is meaningful to draw associations between data values at adjacent grid positions considering that so many natural phenomena are continuous rather than discrete.

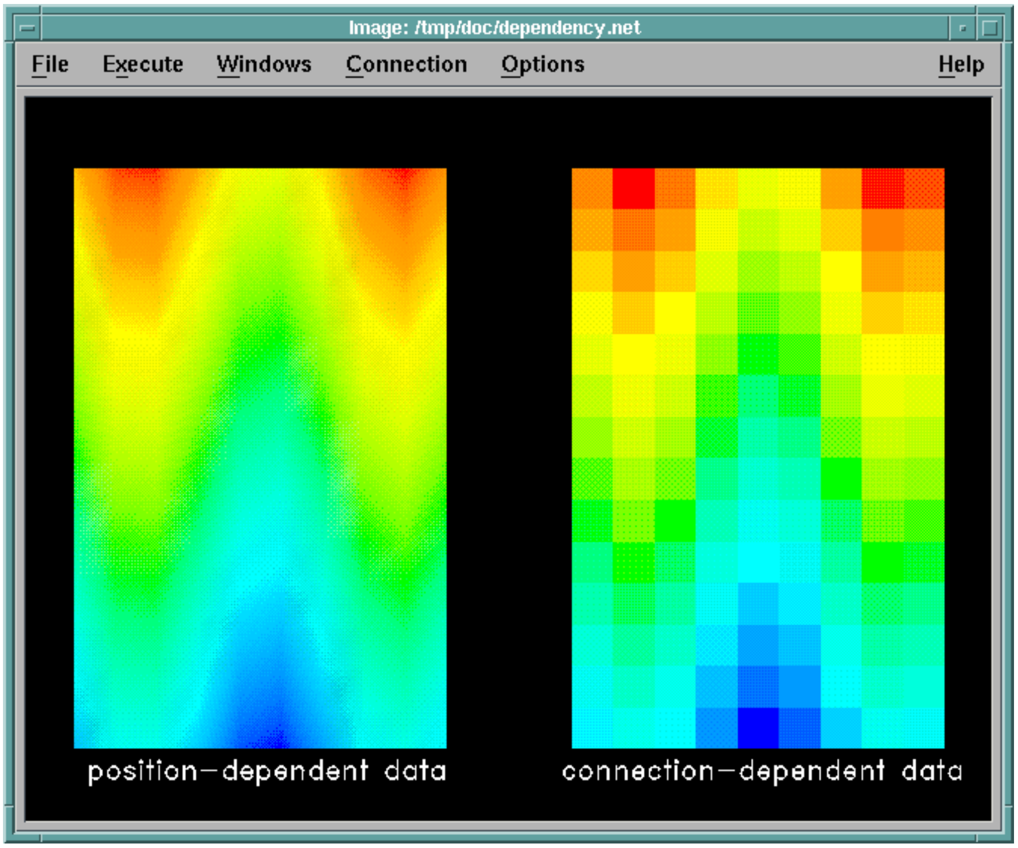


Figure 1. Examples of Data Dependency

We assume that the grasses are free to spread across the area and the wind is free to blow in any direction over the area.

Previously, we assumed that samples were measured at the center of each grid square; that is, the botanist used *quad* connections to associate sets of four positions into 4-sided elements, then measured data values at the center of each connection element, yielding connection-dependent data. Now, assume that the botanist measures temperature values at each grid *position*. Temperature would then be position-dependent data. It's perfectly acceptable to have both kinds of data in the same data set. We will see how this works when we discuss *Fields*.

Assume that the first grid position (sampling point) lies precisely at the position coordinate [x=0,y=0]. We take a measurement and record the value. Then we measure the temperature at [x=1,y=0]. Later, we ask, what was the temperature at [x=0.5,y=0]? Quite honestly, we do not know, because our sampling resolution was not fine enough for us to give a definitive answer. However, if we make the assumption (very often, a perfectly reasonable assumption, but not always!) that our grid overlaid a continuous set of values, we can derive the expected data value by interpolation between known values. If we use *line* connections to connect adjacent points, we realize by looking at our mesh that a straight line connects the grid point [x=0,y=0] and [x=1,y=0] and that halfway along this line lies the grid point [x=0.5,y=0]. We can further assume that the data value at this midpoint is the average of the data values at known sample points bordering this location. By linear interpolation, we calculate a reasonable value for the temperature at [x=0.5,y=0].

We need to define polygonal connections over the 2-D grid if we wish to find the value at the point [x=0.2,y=0.7]. With *line* connections between adjacent pairs of grid points, we can only reasonably perform interpolations along those linear boundaries but not into the middle of our grid elements. By defining areas bounded by three or more points, we can perform interpolation across the area (the polygon surface) using weighting functions that take into account the data values at all points surrounding the area. In fact, this is the same process used by an image-rendering program: it interpolates from known values (at the vertices) across the faces of polygons and computes the appropriate color at all visible points on the surface, at the resolution allowed by the output device (digital file, computer monitor, etc.).

Identifying Connections

In Data Explorer, we identify connections in the following way. List the sample point location vertices in any order: that list is called the “positions” as we discussed above. Consider each point in the positions list to have an ordinal number, starting at 0 for the first point in the list (these ordinal numbers are not explicitly listed in a Data Explorer file). A connection is denoted by a “list of lists” of numbers in which each entry represents the ordinal values of the points that are to be connected, listed in the order they are to be connected. So for example, if the first point in the positions list is “0.0 0.0” and the second point is “1.0 0.0”, we denote a *line* connection between these two points by “0 1”, indicating that a line joins point 0 (first point in the positions list) to point 1 (the second point in the list).

As mentioned above, a *triangle* connection must reference three positions and a *quad* references four positions. For complete examples of position and connection lists, see Chapter 3, “Understanding the Data Model” on page 15.

As a direct extension of this concept, when we define volumetric elements like *cubes* and *tetrahedra*, we can perform 3-dimensional interpolation and derive a reasonable data value for any point in a sample volume. The good news about all of this interpolation is that Data Explorer already knows how to do the necessary calculations. As a researcher, your job is to define your data space to Data Explorer—its positions, connections, and data-dependency—but you do not have to worry about the details of how the interpolation is actually performed.

The connections list is optional if it makes no sense to connect your sample points; for example, if you are studying gas molecules, there may be no meaningful interconnecting lines between separate molecules. Nevertheless, you may wish to define “line” connections linking the atoms within each molecule, in order to visualize interatomic bonds or protein backbones; or you may define cubic volumetric elements in the space around the nucleus if you wish to visualize electronic potential fields, for instance.

In any case, you must define a set of connections before you can perform interpolation operations between sampled data values. This is true both for position-dependent data and for connection-dependent data. Once again, positions are discrete points in space, and connections are logical paths between those points representing reasonable interpolation paths between the sampled data values. If you do not have connection information available, you can use the Connect or Regrid modules to create connections for scattered point data.

If you work with regular grids, the “connections” can be defined in a simple way by Data Explorer regardless of the import format you are using. See Chapter 3, "Understanding the Data Model" on page 15 in this Guide and Chapter 5, "Importing Data" on page 61 in *IBM Visualization Data Explorer QuickStart Guide*.

If your work requires irregular grids, you will need to carefully read the section of this manual that describes the format of Data Explorer element types. You may need to write a filter program to convert the connection list output from your finite element program to the format required by Data Explorer before you can import and visualize data sampled on arbitrary structures.

Invalid Data

Sometimes in the process of collecting or analyzing data, certain regions or positions have no data value associated with them. For example, an instrument may have a "data drop-out" or a simulation may (for whatever reason) produce an invalid entry. Of course, if you are explicitly listing your positions or connections, you can simply leave those positions out when you create your data file. However, if you have a regular grid (for which you simply list the origin of the grid and the delta in each dimension), this is not convenient. Data Explorer has a way to easily handle this situation, using "invalid positions" and "invalid connections" components. These components are discussed in Chapter 3, "Understanding the Data Model" on page 15, but briefly, when present in a Field, they instruct any module processing that Field to completely ignore any position or connection identified in that component. For example, an "invalid positions" component may list the integers 0, 15, and 23. This instructs Data Explorer to ignore the positions 0, 15, and 23 (and the data associated with those positions).

You can create these components in a Data Explorer format file (see Appendix B, "Importing Data: File Formats" on page 241) or, often more easily, using the Include module. For example, suppose in your data file drop-outs are indicated with a data value of 9999, while all valid data lies in the range 0–100. Then set the **max** parameter of Include to 9998. Include will then remove or invalidate all of the positions with the value 9999. Note that it is usually preferable to set the **cull** flag of Include to 0 so that the data values are invalidated rather than actually removed (see Include in *IBM Visualization Data Explorer User's Reference*).

All Data Explorer modules know to ignore invalid data. For example, Streamlines will stop when they reach an invalid element, and Statistics will ignore data values associated with invalid elements.

Fields

Given the sets of numbers, "positions," "connections," and "data", we can define a *Field*, as it is called in Data Explorer. The positions identify locations in space, the (optional) connections define logical continuities (interpolation paths) between positions, and the data are the values measured either at each position or within each connection element. Data Explorer calls each of these sets of numbers (positions, connections, data) a Field *component*. Components are represented as arrays of numbers with some auxiliary information specifying *attributes* (e.g., type of dependency). In addition, there are many other types of Field components. The Field is the basic unit of information in Data Explorer, so it is important to understand how to express your data in these terms.

A Field can only have one "positions" and one "connections" component. A Field can have only one component actually named "data", but you may assign names of your choosing to additional components representing other data sets that are also mapped to the same grid. So you can name a “data” component “temperature” and another “wind velocity”, or you can just use the default name “data” if you only have one “data” component.

The “.dx” file format provides the most flexibility for describing data sets to Data Explorer. But many researchers produce fairly straightforward arrays of numbers mapped onto regular or deformed regular grids. If your data are already written out in such a form, you may not need to convert your data files into the native “.dx” file format. Instead, Data Explorer's General Array Importer can read your data directly, given a small “header” file that you create to tell the General Array Importer the name of your data file and its dimensions (see Chapter 5, “Importing Data” on page 61 in *IBM Visualization Data Explorer QuickStart Guide*).

This shorthand description is enough for Data Explorer to convert your data structure into a Field when it reads your raw data file. You will still find it valuable to understand the components of a Field, because once you begin using the Data Explorer visual programming language, you will have direct access to these components. Much of the power and flexibility of the visual programming language is derived from our ability to access and manipulate Field components in a variety of ways.

2.2 Visual Programming: The Basics

The Field description represents a mapping between your actual data sampling space and the Data Explorer graphics system used to make images of that data space. Given such a mapping, the next step is to learn how to visualize your data in meaningful ways. Data Explorer provides both a visual programming language and a text-based scripting language. The scripting language is described in Chapter 10, “Data Explorer Scripting Language” on page 187. The visual programming language uses a graphically oriented editor instead of a traditional text-based editor as in C or Pascal. You will be using this graphical programming environment to generate graphic images as output; this distinction between graphics as program and graphics as output is subtle, but we do not want to confuse the two.

To build a visual program, you physically select, place, and connect functional *modules*; these are represented graphically as labeled rectangular boxes with tabs sticking out of them. Each module can be thought of as a subroutine in a text-based programming system. You can place multiple instances of the same module, analogous to calling a subroutine several times in a program. Modules have inputs and outputs (those little tabs sticking out) just like the arguments and return values in a text language. The inputs and outputs of modules are connected together into a *network*, which in some ways resembles a flow-chart diagram. (Unlike a flow-chart, you cannot loop back a wire to an earlier input in a Data Explorer visual program) Note that many modules have “hidden” tabs for less commonly used parameters. You can expose hidden parameters by using the **Expand** button in the module’s Configuration dialog box.

Generally speaking, you use Data Explorer to visualize your data in the following way. First, bring in the data from a disk file as a Field (the Import module can read in a Data Explorer format file, a General Array Importer file, or netCDF, CDF, or HDF files). Next, run the imported data Field through one or more modules found in the Realization category. Each of these produces a visual object. You may also want to process these Realizations through Transformation modules to modify the visual or other characteristics of an object. Either one or a collection of visual objects is then displayed in an Image window. The Image window provides a number of convenient tools for interactively rotating your visual objects, zooming in for a closer look at them, and so on. There are many different variations of the above scheme: for example, modules like Construct allow you to create simple Fields without having to import data; Structuring category modules permit you to modify Field components in many ways; other types of output are provided so you can write image files to disk, and so on. But the concept of Import-Realize-Transform-Image is the basic and most common approach to using Data Explorer.

So what happens inside a visual program? The Field with its components flows through one module after another. Some modules add new components, others remove or change components. However, an essential point to keep in mind is that unless a module is designed to operate on a specific component, it does not affect any other part of a Field. That is to say, if you feed a Field into a module that does not operate on the “positions” component, then from the output of that module will come a Field with the identical (unchanged) positions component. And that means that another module further “downstream” in the visual program can operate on that “positions” component if need be. This differs in a critical way from traditional languages, which explicitly specify all return values from a function. In Data Explorer, assume that everything that goes into a module comes out (though often changed), whereas in a traditional language, ignoring side-effects (bad programming practice, usually), only those values specifically indicated as return values are returned when the function exits. The descriptions in *IBM Visualization Data Explorer User’s Reference* identify the components that are changed, deleted, or added by each module.

It is also very often useful to “branch” a visual program. Any module input can only have one wire (“tab connection”) attached to it at a time. However, any module output can feed several different module inputs. This allows you to run copies of the same Field through different “subnets” to perform several different operations on it. To see all of these visual outputs in the same scene, you use the Collect module to gather all the “subnet” output wires back together. The single output from Collect (called a *Group*) can be attached to the input of the Image module. The Collect module shares a handy feature with some other modules in that you can easily add new inputs to it if you need more than the two default input tabs.

See Appendix A, “Using Data Explorer: Some Useful Hints” on page 211 for discussions of the following:

- Visualization techniques (including animation, color mapping, and shading).
- Creating good visualization programs for interactive use.
- Creating good visualizations for video.

Chapter 3. Understanding the Data Model

This chapter describes the concepts and terminology of the data model for data stored in the Data Explorer system, whether in memory or on disk.

A complete understanding of this chapter is not required for the effective use of Data Explorer, and the brief discussion of a Field in Chapter 2, "Introduction to Visualization" on page 7 should get you started. However, the more detailed information here is useful when you have specific questions about the data model.

3.1 Introduction to the Data Model

The Data Explorer data model supports various types of simulation and observational data. Data structures that can be represented include:

- Data defined on a regular orthogonal grid
- Data defined on a deformed regular or curvilinear grid
- Data defined on various irregular grids, such as triangular, quadrilateral, and tetrahedral meshes
- Unstructured data with no connections between the data samples.

The data samples can be defined over spaces of any dimensionality, and, independently, can also be connected by primitives of various dimensionalities (allowing, for example, triangular and quadrilateral meshes defined over 2- or 3-dimensional points). The data values can be associated either with the sample points or with the connections between the sample points. Available data types include:

- Real and complex data
- Scalar, vector and tensor data
- Byte, short, integer (signed and unsigned), and floating-point data

Data are stored in the form of *Objects* for use by Data Explorer modules. An Object is a data structure stored in memory that contains an indication of the Object's type, along with additional type-dependent information. The bulk of the data is encapsulated in *Array* Objects.

The data model centers on the notion of a *sampled field*. The next section describes the *Field, Array,* and *Group* Objects that implement sampled fields in Data Explorer. In addition to these basic Object types, other types are used to construct models for rendering (e.g., Transforms, Clipped Objects, Lights, and Cameras). These are described in B.2, "Data Explorer Native Files" on page 244 and in *IBM Visualization Data Explorer Programmer's Reference*.

Data are also stored in permanent file storage in the form of the same Objects. Although Data Explorer supports the creation of Objects from data stored in other file formats (such as netCDF), the Data Explorer file format offers significant additional functionality and flexibility.

Note that the Data Explorer file format is versatile, allowing for future expansion of the capabilities of the system without requiring changes to the file format. It is possible to represent data types in a Data Explorer file that cannot be processed by the current version of Data Explorer. For example, in the current release of Data Explorer, only single-precision floating-point positions are universally supported. Also, most modules support only 1-, 2-, or 3-dimensional positions.

3.2 Object Types

Field, Array, and Group Objects implement sampled fields in Data Explorer. Additional Object types, used to construct models for rendering, are described in Appendix B, “Importing Data: File Formats” on page 241.

Fields

Field Objects are the fundamental Objects in the Data Explorer data model. A Field represents a mapping from some *domain* to some *data space*. The domain of the mapping is specified by a set of *positions* and (generally) a set of *connections* that allow interpolation of data values for points in the domain between specified positions. The mapping at all points in the domain is represented implicitly by specifying data that are dependent on (located at) the sample points or on the connections between the sample points (cell-centered data).

This simple abstraction is sufficient for representing a wide range of things. For example, you can describe 3-dimensional volumetric data, whose domain is the region specified by the positions, and whose data space is the value associated with each position. Two-dimensional images have a domain that is the set of pixel locations, and a data space that consists of the pixel color. Two-dimensional surfaces imbedded in 3-space (that is, traditional graphical models) can have a domain that is the set of positions on the surface, and a data space that is, for example, the set of data values on that surface.

If the data are dependent on the given positions, then a data value at a point other than those given is found by interpolation within the connection in which the point resides. If the data is dependent on connections, then the data value is assumed to be constant within each connection. If no connections are specified, then there is no implied information about data values at positions other than those given.

The information in a Field is represented by some number of named *components*. Each component has a *value,* that is an Object. In general, components are *Array* Objects (described in more detail in the next section). For example, the “positions” component is an Array specifying the set of sample points; the “connections” component is an Array specifying a means to interpolate between the positions; and the “data” component is an Array specifying the data values.

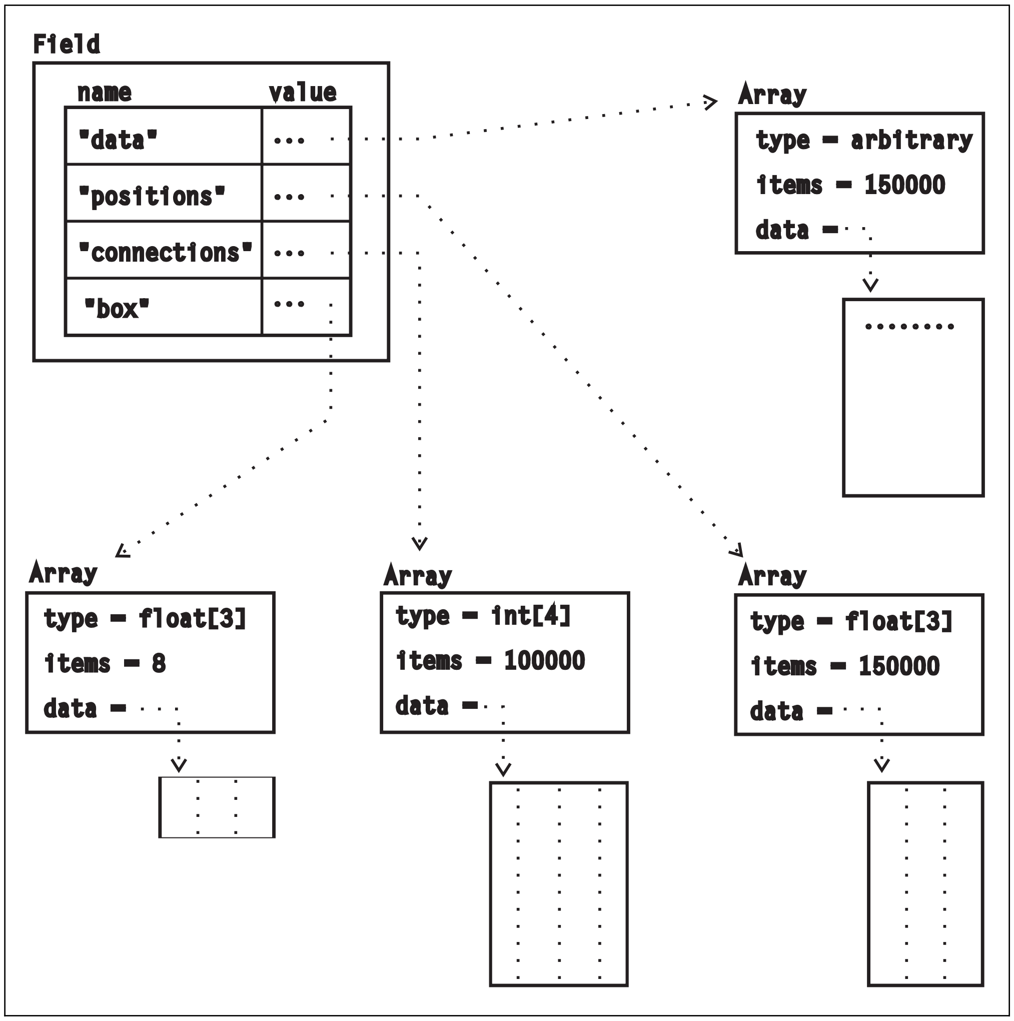


Figure 2. Example of a Field Object

Figure 2 shows an example of a Field Object with four components. The “data” component specifies the user’s data as an Array of data of arbitrary type (e.g., integer), which is dependent on (i.e., in one-to-one correspondence with) the “positions” component; the “positions” component specifies the sample points as an Array of 3-dimensional vectors; the “connections” component specifies a set of tetrahedra as vectors of four integers that refer to the “positions” component; and the “box” component lists the eight points that define the bounding box of the positions (i.e., of the Field itself). A complete list of defined component types is given in “Standard Components” on page 19.

Field components (and Objects in general) can have *attributes* associated with them. For example, the “dep” attribute of a component records the dependency of that component on another component; thus the “data” component will have a “dep” attribute of “positions” or “connections,” depending on whether the data are associated with the sample points or with the connections between them. A component can also have a “ref” attribute, indicating that it refers to another component. Typically, the “connections” component has a “ref” attribute of “positions,” signifying that the items in the connections component refer to the positions component. A “connections” component must have an “element type” attribute naming the type of connections, such as “triangles”, “quads”, or “tetrahedra”. A complete list of defined attributes is given in “Standard Attributes” on page 25; the complete list of element types is given in “Connections Component” on page 20.

Note that Fields can share components. This allows, for example, several Fields to share the same positions and connections while having different data, colors, and so on. Figure 3 illustrates two such Fields that share 3-dimensional positions and tetrahedral connections, but each of which has separate (but still both position-dependent) data. The sharing is possible because the Arrays are Objects with a reference count stored in the Array header.

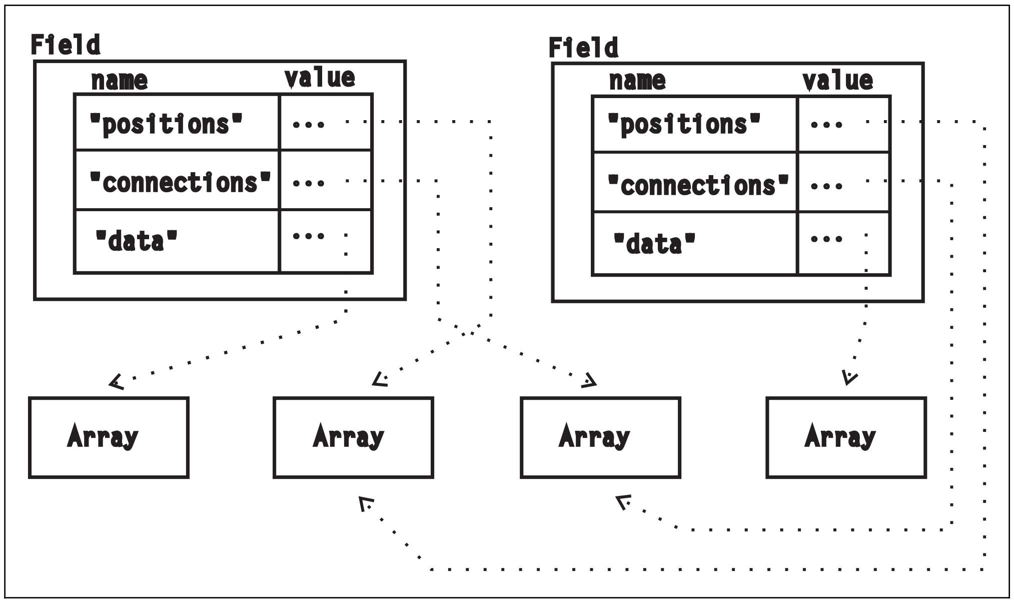


Figure 3. Shared Components among Different Fields

For example, this sharing allows members of a time series, defined on a fixed grid and represented by two Fields, to share positions and connections while each has different data.

In addition, sharing is vital to an efficient implementation of the data flow programming model, in which a module may not modify its inputs. In the example in Figure 3, the first Field might represent the input to a module (e.g., a vector Field), while the second Field might represent the output from a module that computes the length of each vector. The module has constructed a Field with a separate “data” component representing the calculated result, but has not had to copy the portions of the Field that remained the same (positions and connections), because they could be shared between the input and output Fields.

Standard Components

The standard defined Field components are listed in Table 1, and further described in the subsequent paragraphs.

Table 1 (Page 1 of 2). Standard Field Components

Component	**Type**	**Meaning**
“data”	arbitrary	user’s data (dependent variable)
“positions” “invalid positions”	float[*n*] char	*n*-space sample points which sample points are invalid

Table 1 (Page 2 of 2). Standard Field Components

Component	Type	Meaning
“colors”	float[3]	surface or volume colors
“colors”	char	color index (see “color map”)
“color map”	float[3]	color map indexed by “colors” component
“front colors”	float[3]	colors of front of surface
“back colors”	float[3]	colors of back of surface
“opacities”	float	opacity of surface or volume
“opacities”	char	opacity index (see “opacity map”)
“opacity map”	float	opacity map indexed by “opacities” component
“tangents”	float[3]	curve tangent
“normals”	float[3]	curve or surface normal
“binormals”	float[3]	second curve normal
“connections”	int[*k*]	interpolation elements
“invalid connections”	char	which interpolation elements are invalid
“faces”	int	faces described as a collection of loops
“loops”	int	loops described as a series of edges
“edges”	int	edges described as a series of points
“pick”		picks
“paths”		paths
“pickpaths”		pickpaths
“neighbors”	int[*p*]	pointers to connection neighbors
“box”	float[2^n]	2^n corners of a bounding box
“data statistics”		statistics for data component

Positions Component: The “positions” component is an Array Object specifying a set of *n*-dimensional positions. For data on a grid with regular positions, the positions can be encoded compactly by *Regular* and *Product* Arrays, which are described in “Arrays” on page 28.

Connections Component: The “connections” component provides a means for interpolating data values between the positions. Each item of the “connections” Array describes an *interpolation element* such as a line, triangle, tetrahedron or cube. The vertices of each such interpolation element are specified by one Array item consisting of a list of indices into the “positions” Array, one index per vertex of the interpolation element. (Position index numbers begin at 0.)

The type of the interpolation elements is specified by the “element type” attribute of the “connections” component. Two open-ended series of element types are currently defined: the *n*-dimensional simplexes, and the *n*-dimensional cuboids.

The *n*-dimensional simplexes are represented by “connections” components with “element type” attributes of “triangles” (2-D) or “tetrahedra” (3-D). Each item of such a “connections” component is a list of *n+1* integer indices referring to items in the “positions” component representing the *n+1* vertices of an *n*-dimensional simplex. These vertices are ordered as illustrated in Figure 4 on page 21. For tetrahedra, the parity of all tetrahedra in a given Field must be consistent. Figure 4 on page 21 illustrates the two possible parities for tetrahedra. In addition, for triangles there is a convention for which face is the front (using the right-hand rule).

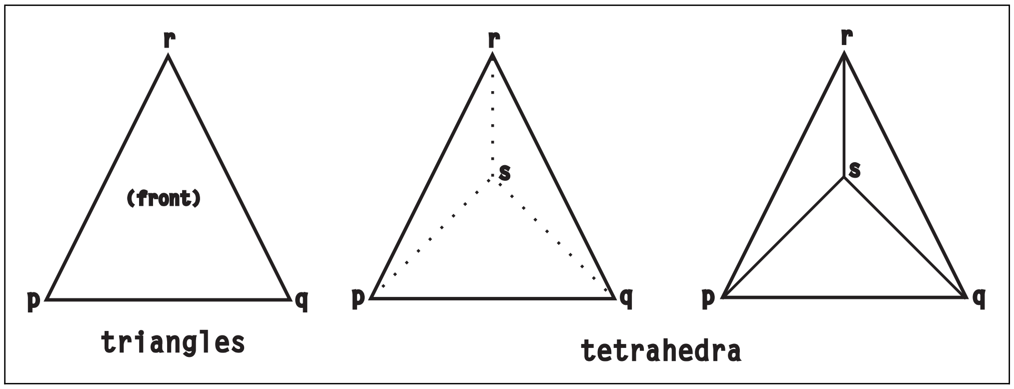


Figure 4. Order of Vertices in Triangles and Tetrahedra. In the tetrahedron at right, **s** *is the point nearest the viewer; at center, the point furthest from the viewer.*

The *n*-dimensional cuboids are represented by “connections” components with “element type” attributes of “lines” (1D), “quads” (2-D), “cubes” (3-D), “cubes4D”, and so on in the format “cubes*n*D”, where *n* represents the number of dimensions. Each item of such a “connections” component is a list of 2^n integer indices referring to items in the “positions” component representing the 2^n vertices of an *n*-dimensional cuboid. The ordering of these vertices is illustrated in Figure 5. For cubes, the parity of all cubes in a given Field must be consistent. In addition, for quads there is a convention that determines the front face.

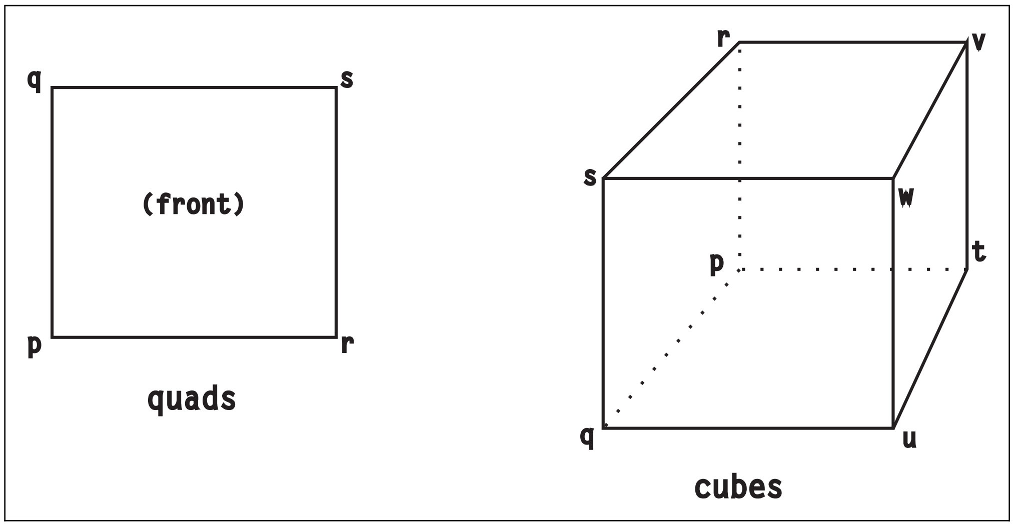


Figure 5. Order of Vertices in Quads and Cuboids

Note: Figure 5 does not indicate the correspondence between the edges of the cubes or quads and the spatial dimensions. For example, the cubes or quads can be “irregular,” in which case the positions of each vertex are specified explicitly. Regular “positions” components can specify an arbitrary correspondence between the spatial dimensions and the edges of the cube, as illustrated in Figure 10 on page 30.

For data on grids with regular connections, the connections can be encoded compactly by *Path* and *Mesh* Arrays, which are described in “Arrays” on page 28 and in more detail in Appendix B, “Importing Data: File Formats” on page 241.

Figure 6 on page 22 illustrates the various types of grids formed with different kinds of “positions” and “connections” components.

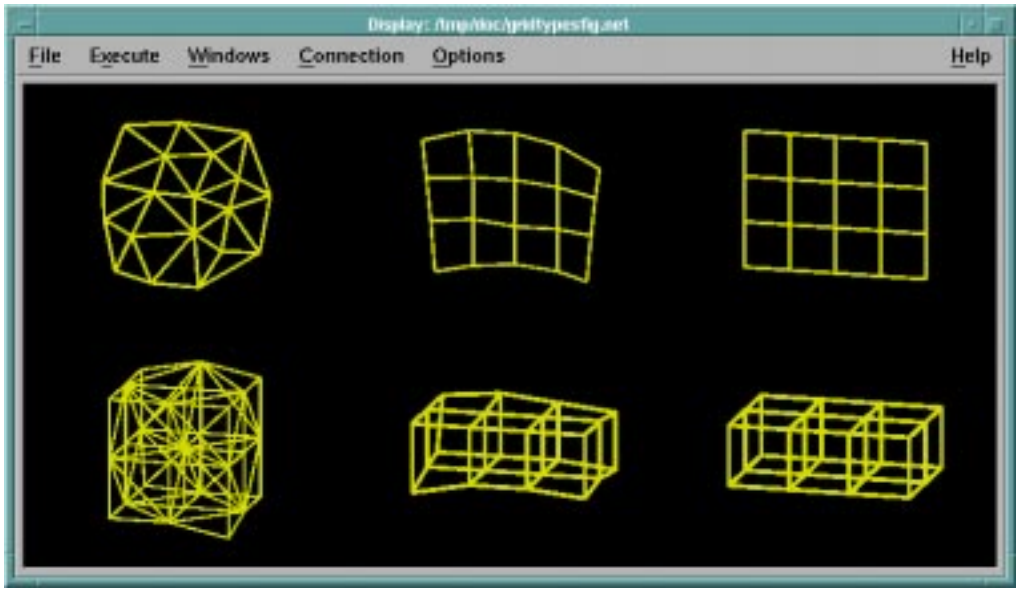


Figure 6. Examples of Grid Types. The three grids in the top row represent surfaces; those in the bottom row, volumes. Reading from left to right, the three grid types are: irregular (irregular positions, irregular connections), deformed regular (irregular positions, regular connections), and regular (regular positions, regular connections).

Data Component: The "data" component stores the user's data values. The data values can be position- or connection-dependent, as specified by the value of the "dep" attribute (described in "Standard Attributes" on page 25) of the "data" component. If the values are position-dependent, then the "connections" component supplies a means of interpolating data values between the samples. If the values are connection-dependent, the data value is constant for each interpolation element. Data can also be dependent on "faces" or "polylines" (see "Edges and Polylines" on page 25), in which case the data is constant for either the face or the polyline.

The data are in one-to-one correspondence with the component upon which they are dependent. This means that they are specified in the same order as the items in the corresponding component. If that component is specified in a compact form, its contents are ordered as described in "Arrays" on page 28.

Colors, Front Colors, and Back Colors Components: The "colors", "front colors", and "back colors" components are a means of specifying another, specialized type of dependent data. Specifically, the renderer requires that each object in a scene have at least one of these components. The "front colors" and "back colors" components specify colors to be associated with the front and back sides, respectively, of a surface. The "colors" component specifies colors to be associated with both sides of a surface, or with a volume. If only "front colors" or only "back colors" are specified, then only the "front" or "back" sides, respectively, of the polygons are rendered. If present, "front colors" or "back colors" override the specification of the "colors" component.

Each item of a color component Array consists of three floating-point numbers specifying the red, green, and blue component of a color respectively. The color components can use the entire floating-point range, but by convention the range from 0 to 1 is mapped onto the available range of the output device. Like the "data" component, the color components can be position- or connection-dependent.

The front of a triangle is defined to be the side such that the path traversing the vertices in the order that they are listed for the triangle appears to go counterclockwise. The front of a quad is the side from which the vertex numbering appears (Figure 5 on page 21).

The interpretation of colors differs between surfaces and volumes. For surfaces, the color values in the range from 0 to 1 are mapped onto the range of colors values possible for the display. For volumes, the interpretation of colors follows the "dense emitter" model described in the next section.

Opacities Component: The "opacities" component plays a role similar to that of colors components, except that it specifies a floating-point opacity for rendering. Its interpretation differs depending on whether the connections represent a surface (triangles or quads), or a volume (tetrahedra, cubes, and so on). In the surface case, the opacity is a number from 0 to 1 specifying the opacity of the surface. In the volume case, the opacity represents the instantaneous attenuation of light per unit distance traveled. Like the colors components, it can be position- or connection-dependent.

The interpretation of "colors" and "opacities" differs between surfaces and volumes. For surfaces, a surface of color c_f and opacity o is combined with the color c_b of the objects behind it resulting in a combined color $c_f o + c_b(1 - o)$.

For volumes, the "dense emitter" model is used, in which the opacity represents the instantaneous rate of absorption of light passing through the volume per unit thickness, and the color represents the instantaneous rate of light emission per unit thickness. If $c(z)$ represents the color of the object at z and $o(z)$ represents its opacity at z, then the total color of a ray passing through the volume is given by:

$$c = \int_{-\infty}^{\infty} c(z) \exp\left(-\int_{-\infty}^{z} o(\zeta) d\zeta \right) dz$$

Color Map and Opacity Map Components: There is an alternative to having the "colors" component and "opacities" component explicitly list the color and opacity associated with each position or connection element. If each element of the "colors" and "opacities" components is a single byte, then it is interpreted as an index into the "color map" or "opacity map" component. The "color map" component is a 256-element table of three floating-point values (representing red, green, and blue, typically with values between 0 and 1). The "opacity map" component is a 256-element table of floating-point values between 0 and 1.

Invalid Positions and Invalid Connections Components: The "invalid positions" and "invalid connections" components allow positions or connections to be marked as not having valid data. This is useful, for example, for observational data defined on a regular grid but with occasional missing observations, or for simulation data defined on a regular grid but with a "hole" covered by another grid, perhaps at a higher resolution. The "invalid positions" component can be an Array of bytes or unsigned bytes, one for each position, where the component is dependent on positions (i.e., has a "dep" attribute of "positions."). In that case, the value 1 indicates that the corresponding position is invalid, and 0 that the corresponding position is valid. Alternatively, the "invalid positions" component can be an Array of signed or unsigned integers, where the component references the positions (i.e., has a "ref" attribute of "positions"). In that case, the component contains a list of the indices of the invalid positions. The first method is more space-conserving when there are a large number of invalid elements; the second, when there are a relatively small number. See “Compute” on page 86 in *IBM Visualization Data Explorer User’s Reference* for a way to convert from the “ref” type to the “dep” type.

The “invalid connections,” “invalid faces,” and “invalid polylines” components can be defined in an analogous way.

Tangents, Normals, and Binormals Components: The “normals” component is used to specify a local surface normal for rendering purposes. The “tangents”, “normals” and “binormals” components specify a local reference frame on a path; this is useful, for example, for twisted-ribbon representations of streamlines.

Normals are used for, among other things, surface shading. By convention, the normals are expected to point out from the front of a surface, as defined in “Colors, Front Colors, and Back Colors Components” on page 22. Normals are expected to have unit length.

Neighbors Component: The “neighbors” component represents information about the neighbors of each connection element. The number of items in this component must match the number of items in the “connections” component. The number of entries in each item must match the number of faces (for 3-D) or edges (for 2-D) in the connection element. For example, each item in the “neighbors” component for triangle connections has three entries, while each item in the “neighbors” component for tetrahedral connections has four entries.

For simplexes in *n* dimensions (for example, triangles and tetrahedra), each item of the neighbors Array consists of *n+1* integer indices into the connections Array identifying the *n+1* neighbors of the simplex; the *i*th of the *n+1* indices corresponds to the face opposite the *i*th vertex of the simplex. For quads, cubes, and so on, each item of the neighbors Array contains 2*n* integer indices into the connections Array identifying the 2*n* neighbors of the polyhedron. The pointers are in the order $-x_1 +x_1 -x_2 +x_2 \ldots -x_n +x_n$, meaning that the first index points to the neighbor in the $-x_1$ direction, the second to the neighbor in the $+x_1$ direction, and so on, where the x_n dimension varies fastest in the representation of the point indices in the interpolation element. Faces without neighbors are indicated by an index of −1.

Box Component: The “box” component consists of 2^n *n*-dimensional points, where *n* is the dimensionality of the positions component, identifying the corners of a bounding box that contains the positions of this Field.

Data Statistics Component: The “data statistics” component contains statistics of the “data” component. The information in this component should be accessed using the Statistics module or the **`DXStatistics()`** function, as the exact contents are undefined. If DXStatistics is called on other components (e.g., “positions”), an analogous component (in this case “positions statistics”) will be created.

Faces, Loops, and Edges Components: The “faces”, “loops”, and “edges” components are used for special-purpose applications, such as fonts or geometric models. The “faces” component represents a set of faces, each described as a set of loops. Each entry in the face Array is a single integer index into the “loops” Array identifying the first of a consecutive set of loops for this face. The loops are listed in order of the faces they are associated with, so that the list of loops for face *i* ends in the loops Array just before the first loop for face *i+1*. Each entry in the “loops” Array is a single integer index into the “edges” Array, identifying the first of a consecutive set of edges for this loop. The edges are listed in order of the loops they are associated with, so that the list of edges for loop *i* in the edges Array ends just before the first edge for loop *i+1*. Each entry in the edges Array is a single integer index into the "points" Array identifying one vertex of an edge; the other vertex of an edge is the next entry in the "edges" Array, except that the last edge in a loop that connects the last point to the first point is not listed explicitly. This is illustrated in Figure 7 on page 26.

It is assumed that the first loop for each face is the enclosing loop, and that subsequent loops, if any, are holes in the face. If this is not true, then set the DX_NESTED_LOOPS environment variable (see "Other Environment Variables" on page 292). However, setting this environment variable will cause a decrease in performance when processing faces, loops, and edges data.

Edges and Polylines: "Polylines" are a way of collecting a set of line segments into a single object with which data can be associated. They are implemented much as faces, loops, and edges are (see above). An "edges" component contains the indices of the vertices along polylines. A "polylines" component contains the indices of the first element in the "edges" component of each polyline sequence. In other words, the i^{th} element of the "polylines" component is the index in the "edges" component at which the sequence of vertex indices of polyline *i* starts. The sequence corresponding to polyline[*i*] continues to the beginning of the next polyline sequence (or to the end of the "edges" component). Polyline data may be dependent on either "polylines" or "positions".

Pokes, Picks, and Pick Paths Components: The pokes, picks, and pick paths components are created as part of the picking process, as implemented by the Pick tool. A user writing a module that uses the pick structure output by the Pick tool is expected to use the pick structure manipulation routines (as described in *IBM Visualization Data Explorer Programmer's Reference*) rather than accessing the pick structure directly. The contents of the pick structure are not defined.

Standard Attributes

The standard defined attributes are listed in Table 2 and Table 3, and are further described in the subsequent paragraphs. Attributes associated with *rendering properties* are described under the Display module in Chapter 1, "Data Explorer Tools" on page 1 in *IBM Visualization Data Explorer User's Reference*.

Table 2. Component attributes

Attribute	Meaning
"dep"	component that this component depends on
"ref"	component that this component refers to
"der"	component that this component is derived from
"element type"	interpolation method for connections component

Table 3 (Page 1 of 2). Object attributes

Object	Relevant module
label	Plot
scatter	Plot
mark	Plot

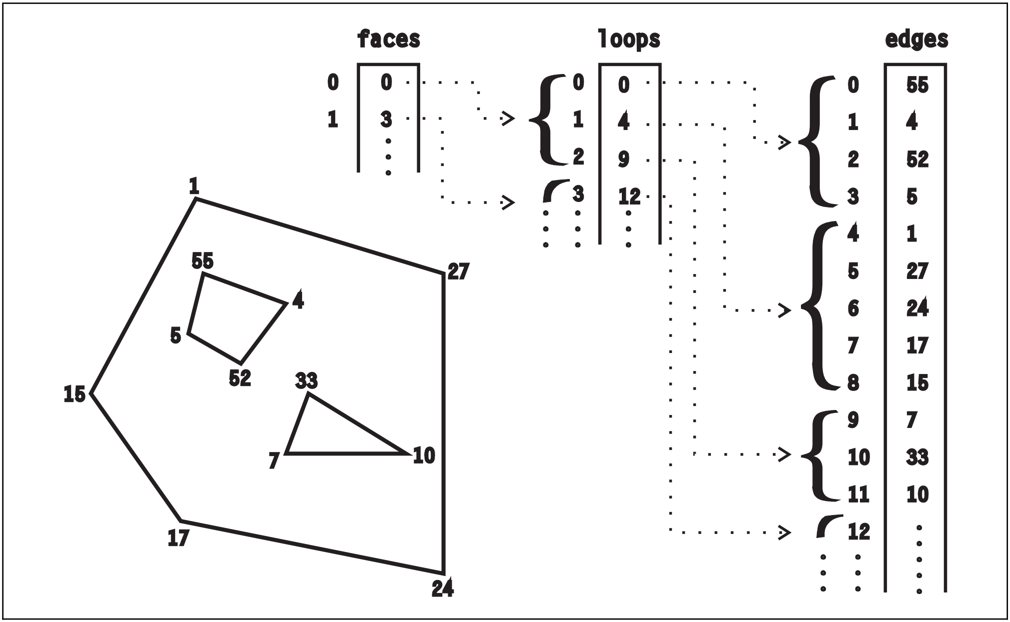


Figure 7. Use of Faces, Loops, and Edge Components

Table 3 (Page 2 of 2). Object attributes	
Object	**Relevant module**
mark every	Plot
mark scale	Plot
fuzz	Display, Render, Image
ambient	Display, Render, Image
diffuse	Display, Render, Image
specular	Display, Render, Image
shininess	Display, Render, Image
shade	Display, Render, Image
opacity multiplier	Display, Render, Image
color multiplier	Display, Render, Image
texture	Display, Image
direct color map	Display
cache	Display, Image
rendering mode	Display, Image
rendering approximation	Display, Image
render every	Display, Image
pickable	Pick
marked component	Mark, Unmark

dep Attribute: The “dep” attribute specifies which component the given component depends on. The dependent component is specified by a String Object naming the component it depends on. For example, if the data is position-dependent, it has a “dep” attribute that is a String Object naming the “positions” component. A component with a this attribute is expected to be in a one-to-one correspondence with the component named in the attribute.

ref Attribute: The “ref” attribute specifies which component the given component refers to. The referent component is specified by a String Object naming the component it refers to. For example, the “connections” component generally has a “ref”attribute that is a String Object naming the “positions” component. A component with this attribute consists of indices into the component named in the attribute.

der Attribute: The “der” attribute specifies that a component is derived from another component, and so should be recalculated or deleted when the component it is derived from changes. For example, the “box” component has a “der” attribute that is a String Object naming the “positions” component.

element type Attribute: The “element type” attribute is an attribute of the “connections” component. It is a String Object naming the type of the interpolation primitives. See “Connections Component” on page 20 for a list of the possible values of the “element type” attribute.

label, scatter, mark, mark every, mark scale Attributes Specifies characteristics of a plotted line. Affects the behavior of the Plot module. See “Plot” on page 237 in *IBM Visualization Data Explorer User’s Reference* for more information.

fuzz, ambient, diffuse, specular, shininess, shade Attributes Specifies various rendering characteristics of an object. Affects the behavior of the rendering modules (Display, Render, and Image). See “Display” on page 109 in *IBM Visualization Data Explorer User’s Reference* for more information.

opacity multiplier, color multiplier Attributes Specifies opacity and color values for volume rendering. Affects the behavior of the rendering modules (Display, Render, and Image). See “Display” on page 109 in *IBM Visualization Data Explorer User’s Reference* for more information.

texture Attribute Specifies a texture map which is to be applied to an object. Affects the behavior of Display and Image. See “Display” on page 109 in *IBM Visualization Data Explorer User’s Reference* for more information.

direct color map Attribute Specifies whether or not a direct color map should be used when displaying images. Affects the behavior of the Display module. See “Display” on page 109 in *IBM Visualization Data Explorer User’s Reference* for more information.

cache Attribute Specifies whether the rendered image should be cached. Affects the behavior of Display and Image modules. See “Display” on page 109 in *IBM Visualization Data Explorer User’s Reference* for more information.

rendering mode Attribute Specifies the rendering mode to be either hardware or software. Affects the behavior of Display and Image modules. See “Display” on page 109 in *IBM Visualization Data Explorer User’s Reference* for more information.

rendering approximation, render every Attributes Specify hardware rendering characteristics for an object. Affect the behavior of Display and Image modules. See “Display” on page 109 in *IBM Visualization Data Explorer User's Reference* for more information.

pickable Attribute Specifies whether or not an object should be pickable. See “Pick” on page 234 in *IBM Visualization Data Explorer User's Reference* for more information.

marked component Attribute Specify which component in an object was previously marked. This attribute is set by the Mark module; affects the behavior of the Unmark module. See “Mark” on page 214 and “Unmark” on page 358 in *IBM Visualization Data Explorer User's Reference* for more information.

Arrays

Array Objects hold the actual data, positions, connections, and so on. An Array consists of some number of *items* numbered consecutively starting at 0. Each item has a *type, category, rank* and *shape,* defined as follows:

Type Types include **`double`**, **`float`**, **`int`**, **`uint`**, **`short`**, **`ushort`**, **`byte`**, **`ubyte`**, and **`string`**. (For example, **`byte`** is signed byte and **`ubyte`** is unsigned byte.)

Category A category can be **`real`** or **`complex`**.

Rank Rank 0 corresponds to scalars, rank 1 to vectors, rank 2 to matrices or rank-2 tensors; higher ranks correspond to higher-order tensors.

Shape The shape is defined as the list of dimensions of the structure. For rank-0 items (scalars), there is no shape. For rank-1 structures (vectors), the shape is a single number corresponding to the number of dimensions. For rank-2 structures, shape is two numbers, and so on.

The following are examples of these classifications:

- Three-dimensional points have type **`float`** or **`double`**, category **`real`**, rank 1, and shape 3.
- Two-vectors typically have type **`float`**. They are category **`real`**, rank 1, and shape 2. Three-vectors are shape 3.
- Tetrahedra have type **`int`**, category **`real`**, rank 1, and shape 4.
- Scalar values typically have type **`int`** or **`float`**. They are category **`real`** and rank 0, with no shape.
- Strain tensors typically have type **`float`** or **`double`**. They are category **`real`**, rank 2, and shape $3{\times}3$.

Data Explorer uses six types of Array: *Irregular Arrays* and five types of compact Array: *Regular*, *Product*, *Path*, *Mesh*, and *Constant*.

The Array types are discussed in the following sections.

Irregular Arrays

The most general way to specify the contents (item values) of an Array is to list the values; this is called *irregular* data. An example of such an Array Object is illustrated in Figure 8 on page 29.

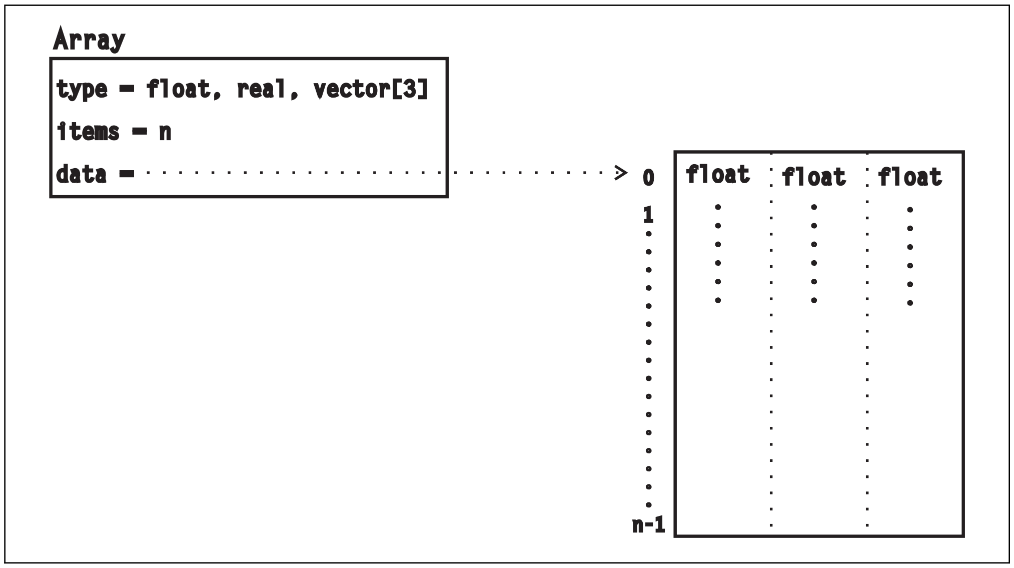


Figure 8. Example of an Irregular Array

Regular Array

A set of *n*-dimensional points lying on a line in *n*-space with a constant *n*-dimensional delta between them, represents, for example, one edge of a grid of regular positions. Regular Arrays are frequently combined as the terms of a Product Array. An example of a Regular Array is illustrated in Figure 9.

This example represents (in compact form) the same information as the following irregular Array:

$$[x_o, \quad y_o]$$
$$[x_o + x_d, \quad y_o + y_d]$$
$$[x_o + 2x_d, \quad y_o + 2y_d]$$
$$\vdots$$
$$[x_o + (n-1)x_d, \quad y_o + (n-1)y_d]$$

```
type   = float, real, vector[2]
items  = n
origin = [xo, yo]
delta  = [xd, yd]
```

Figure 9. Example of a Regular Array

Product Array

Encodes multidimensional positional regularity. It is the set of points obtained by summing one point from each of the terms of the product in all possible combinations. For example, the product of a set of Regular Arrays is a regular grid whose basis vectors are the deltas of the Regular Arrays that are the terms of the product, and whose origin is the sum of the origins of the terms. An example of a Product Array Object is illustrated in Figure 10 on page 30. A Product Array can have terms that are Regular Arrays, irregular Arrays, or any combination of Regular and irregular Arrays.

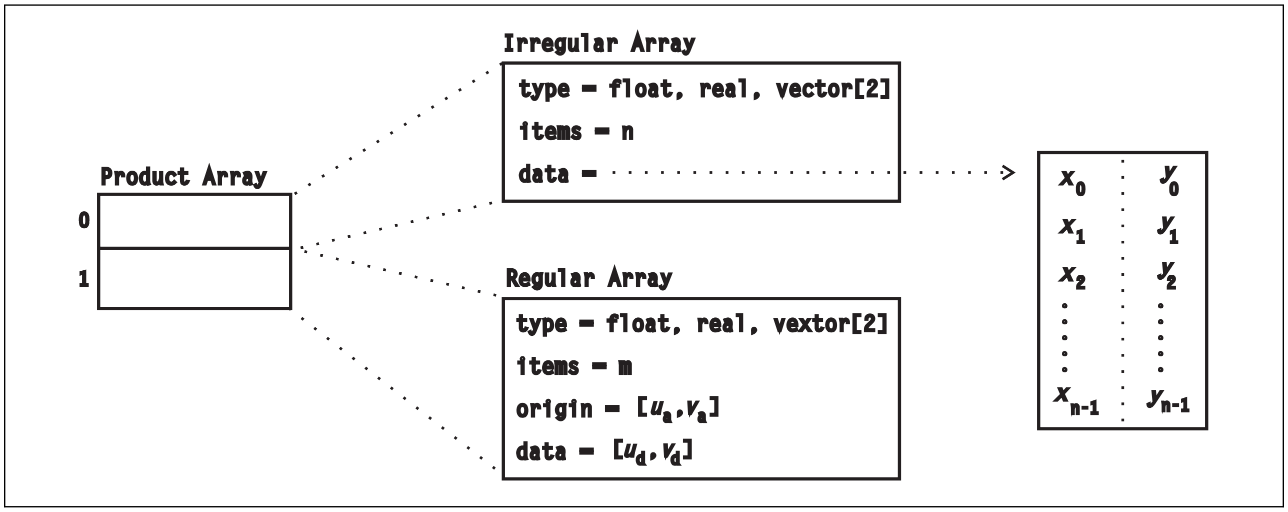


Figure 10. Example of a Product Array

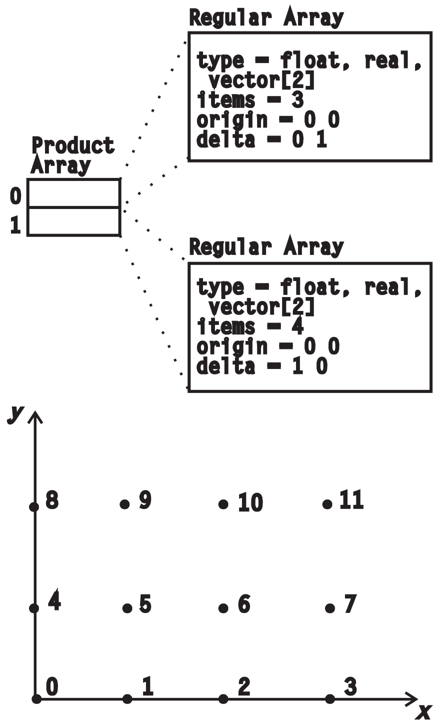
The example in Figure 10 represents (in compact form) the same information as the following irregular Array:

$$
\begin{array}{l}
[x_o + u_o, \quad y_o + v_o] \\
[x_o + u_o + u_d, \quad y_o + v_o + v_d] \\
[x_o + u_o + 2u_d, \quad y_o + v_o + 2v_d] \\
\vdots \\
[x_1 + u_o, \quad y_1 + v_o] \\
[x_1 + u_o + u_d, \quad y_1 + v_o + v_d] \\
\vdots \\
[x_{n-1} + u_o + (m-1)u_d, \quad y_{n-1} + v_o + (m-1)v_d]
\end{array}
$$

An important special case of the more general Product Array Object is the *n*-dimensional geometrically regular grid. Figure 11 on page 31 is an example that shows two ways to describe a Product Array composed of two Regular Arrays.

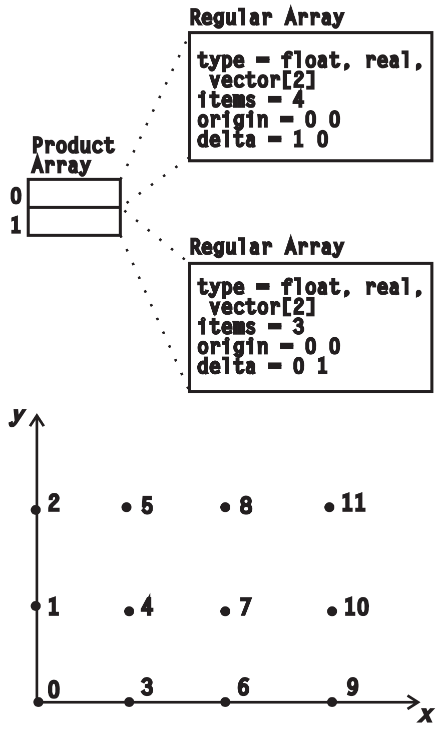
The order of the specification of the counts and deltas implicitly creates a list of positions.

x is the fastest-varying dimension.



This represents (in compact form) the same information as the following irregular Array:
[0 0]
[1 0]
[2 0]
[3 0]
[0 1]
[1 1]
[2 1]
⋮

y is the fastest-varying dimension.



This represents (in compact form) the same information as the following irregular Array:
[0 0]
[0 1]
[0 2]
[1 0]
[1 1]
[1 2]
[2 0]
⋮

Figure 11. Product Array of Two Regular Arrays

Path Array

Encodes linear regularity of connections. It is a set of *n-1* line segments joining *n* points, where the *i*th line segment joins points *i* and *i+1*. Path Arrays are frequently combined as the terms of a Mesh Array. An example of a Path Array is illustrated in Figure 12.

```
type  = int, real, vector[2]
items = n
```

Figure 12. Example of a Path Array

This example represents (in compact form) the same information as the following irregular Array:

$[0\ \ 1]$
$[1\ \ 2]$
$[2\ \ 3]$
$\vdots$
$[n-2\ \ n-1]$

Mesh Array

Encodes multidimensional regularity of connections. It is a product of connection Arrays. The product is a set of interpolation elements where the product has one interpolation element for each pair of interpolation elements in the two multiplicands, and the number of sample points in each interpolation element is the product of the number of sample points in each of the multiplicands' interpolation elements. An example of a Mesh Array is illustrated in Figure 13.

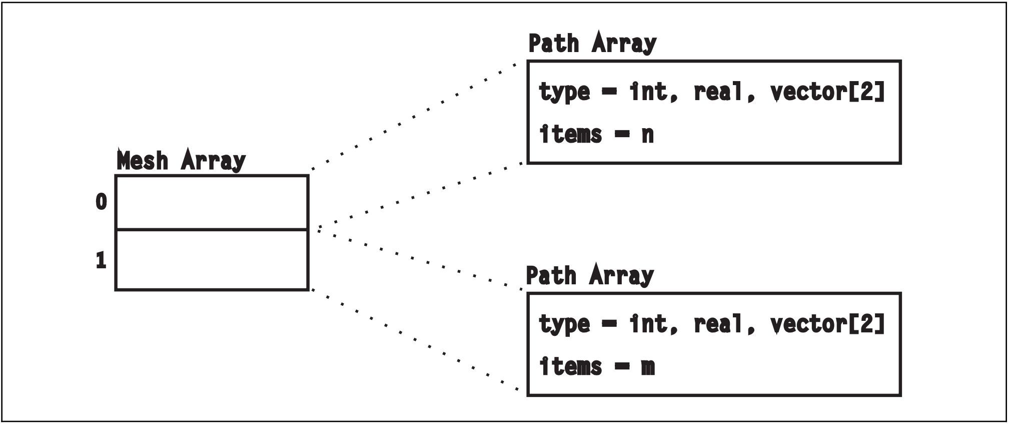
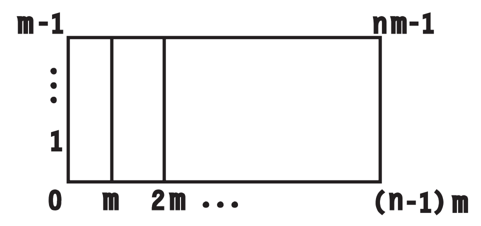


Figure 13. Example of a Mesh Array

This example represents (in compact form) the same information as the following irregular Array:

$[0, 1, m, m+1]$
$[1, 2, m+1, m+2]$
$\vdots$
$[m-2, m-1, 2m-2, 2m-1]$
$[m, m+1, 2m, 2m+1]$
$\vdots$

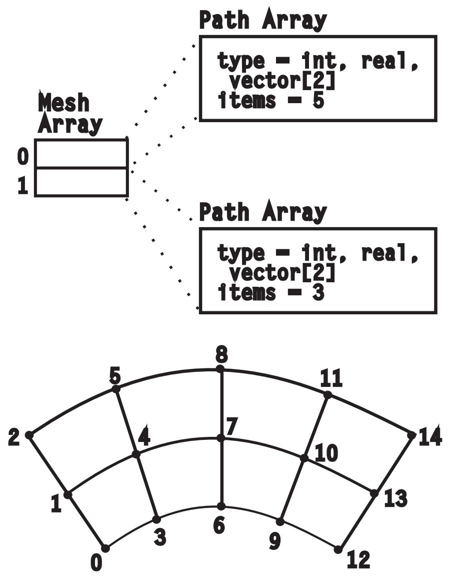


An important special case of the more general Mesh Array Object is the *n*-dimensional cuboidal connections of a regular grid. Figure 14 on page 33 is an example that shows a Mesh Array composed of two Path Arrays.

See Appendix B, "Importing Data: File Formats" on page 241 for a detailed description of how to specify these compact Arrays.

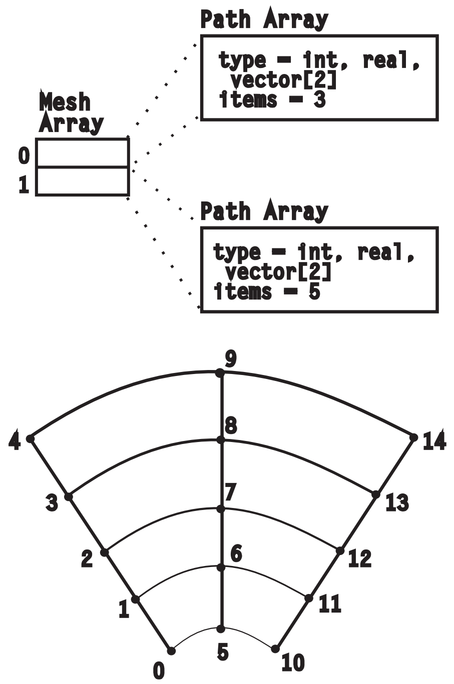
The order of the specification of the counts implicitly creates a list of position indices in the order of "last index varies fastest."

The location in space of each vertex is determined by the value of the position referred to by that index.


Path Array
type = int, real, vector[2] items = 5
Mesh Array
0
1
Path Array
type = int, real, vector[2] items = 3
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

This represents (in compact form) the same information as the following irregular Array:

[0 1 3 4]
[1 2 4 5]
[3 4 6 7]
⋮


Path Array
type = int, real, vector[2] items = 3
Mesh Array
0
1
Path Array
type = int, real, vector[2] items = 5
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

This represents (in compact form) the same information as the following irregular Array:

[0 1 5 6]
[1 2 6 7]
[2 3 7 8]
⋮

Figure 14. Mesh Array of Two Path Arrays (with Regular Connections)

Constant Array

Encodes a set of numbers all with the same value. This can be useful, for example, for specifying the colors associated with an object if the object has a single color. An example of a Constant Array is shown in Figure 15.

```
type   = float, real, vector[3]
items  = n
origin = [xo, yo, zo]
```

Figure 15. Example of a Constant Array

Groups

Fields can be combined into *Groups*. A Group is a collection of *members* that themselves may be Fields or other Groups. A member can be referred to either by name or by index. An example of a Group is given in Figure 16.

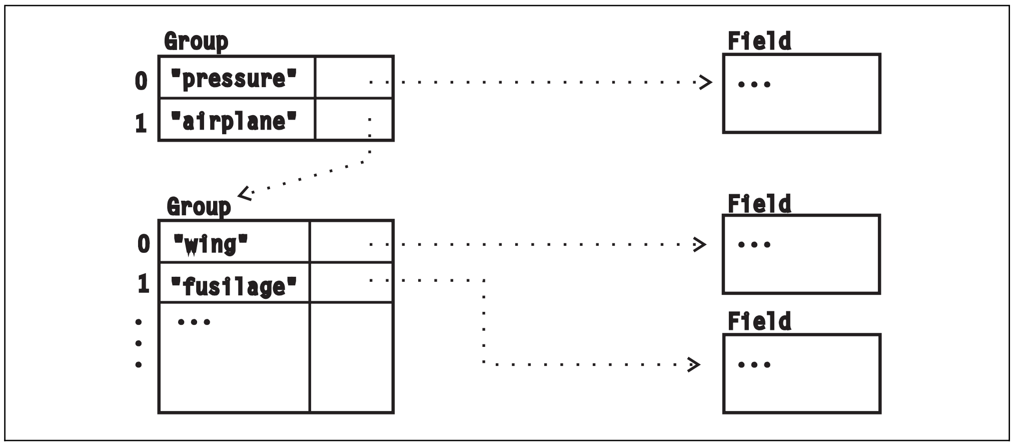


Figure 16. Example of a Group

This example shows a Group describing a visualization scene composed of two parts. The member named “pressure” is a Field of volumetric data to be volume rendered, representing pressure in an airflow around an airplane. The member named “airplane” is a geometric model describing the surface of the airplane. It, in turn, can be a Group, where each member is one of the constituent parts of the airplane, such as “wing,” “fuselage,” and so on. The top-level Group could then be passed into the renderer to produce an image of the airplane combined with a volume rendering of the pressure Field.

Named Group members can be retrieved by name or by index number; the *n* members of a Group are numbered from 0 to *n-1*. Group members can also be stored by index number without a name, in which case they can be retrieved only by index number. The members of a Group are always numbered consecutively starting at 0, and gaps in the numbering are not allowed.

In addition to generic Groups used to collect related information, there are three subclasses of Group used to combine related Objects with additional semantics: *Multigrid* Groups, *Composite Field* Groups, and *Series* Groups.

Multigrid Groups

It is often necessary to represent a Field as a collection of separate Fields, each with its own grid. For example, this is the case in some kinds of simulations using multiple grids. The data structure used to hold such Fields is a subclass of Group called a *Multigrid* Object. It is the same as a generic Group in most respects, except that it requires all members to be Fields holding data of the same type. The “connections” component of each member must also be of the same type. Grids may be completely disjoint or may overlap. For overlapping grids, the “invalid positions” or “invalid connections” components may be used to define which grid is valid in a particular region.

Composite Field Groups

A Composite Field is another kind of Group that is treated as a single entity. For example, parallelism in Data Explorer is achieved by explicitly partitioning Fields into abutting, spatially disjoint primitive Fields. Positions on the boundaries must be replicated identically. Like Multigrid Groups, all members must have the same type of data and the same type of connections.

Series Groups

Series of various types, such as time series, are stored in a subclass of Group called a *Series* Object. A Series Object is the same as a generic Group in most respects, except that it associates a series value, such as a time stamp, with each member. Members are stored in and retrieved from a Series Group by index. Members cannot be retrieved by series value. Fields in a Series Group must all have the same data type and connection element type. Figure 17 shows an example of a Series Group, where the three members have series positions of 1.2, 2.7, and 8.4 respectively.

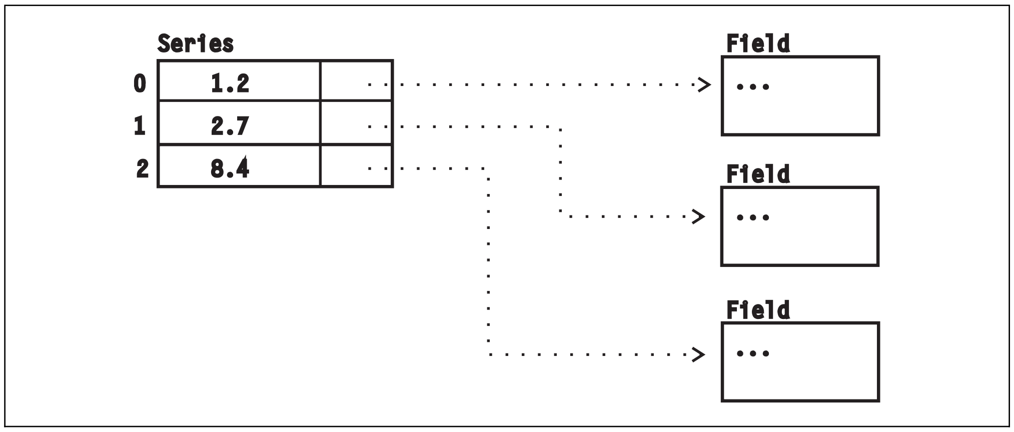


Figure 17. Example of a Series Group

Chapter 4. Data Explorer Execution Model

Data Explorer's execution model is based on the data flow concept. However, features are provided that extend the data flow concept to allow you to create a visual program that could not be supported by simple data flow. For example, there are tools that allow you to explicitly save partial results of a visual program to be used in a subsequent execution. Data Explorer also provides you with various tools to control the flow of execution of your visual programs. Most of these tools are analogous to constructs found in commonly used programming languages. For example, tools are provided that perform the function of IF statements, CASE statements and FOR (or DO) loops. This chapter discusses flow-control tools (with several examples of their use) both individually and in combination.

Although it is not necessary to understand all the details of the Data Explorer execution model in order to build and run visual programs, you may find it helpful as you build visual programs. Other topics in this chapter include caching of intermediate results, conditional execution using the Route and Switch modules, iteration using the Sequencer, simple iteration using the looping modules, preservation of state using the pairs GetLocal/SetLocal or GetGlobal/SetGlobal, creating advanced looping constructs using combinations of tools, asynchronous data sources, and parallelism. The tools that control the flow of execution of a visual program are found in the Data Explorer category **Flow Control**.

4.1 Data Flow

In a true data-flow implementation, all modules are pure functions (i.e., their outputs are fully defined by their inputs). Hence, processes are stateless with no side effects. When a module's inputs are received, it runs, and when finished it distributes its results to modules waiting downstream. Note that in Data Explorer, results are communicated between modules by passing pointers to data objects, not by copying. Of course, when running in distributed mode or when using outboard modules, data must be sent by socket since the processing may occur on another host.

Consider the example illustrated in Figure 18 on page 39.

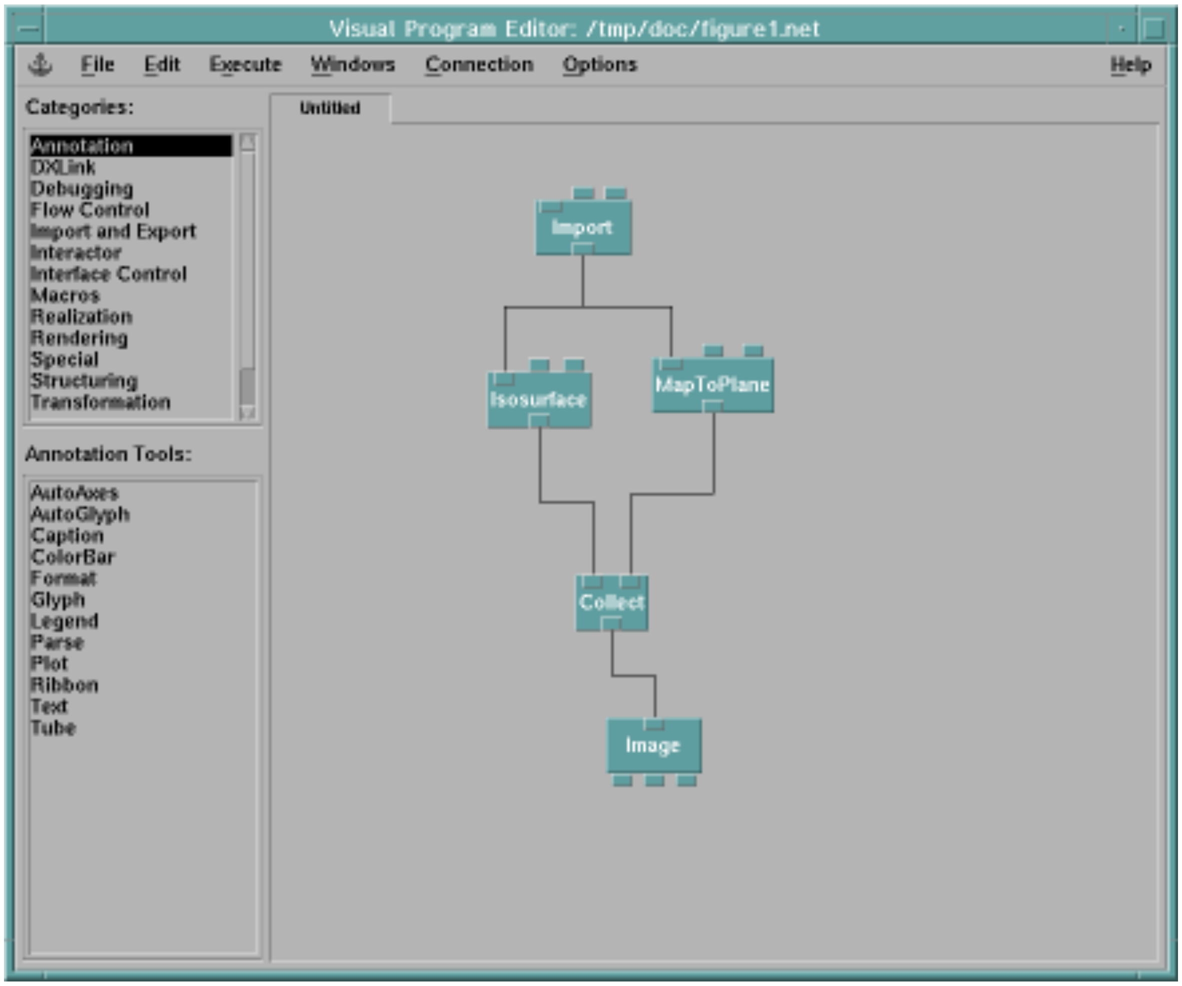


Figure 18. Example 1

The Collect module waits for inputs from the Isosurface and MapToPlane modules. Import would send its results to the waiting Isosurface and MapToPlane modules. In effect, this execution model is entirely data-driven and top-down: the execution of modules is dependent solely on the passage of data through the system.

While this simple data-flow execution model seems a natural mechanism for the execution of visual programs, a closer examination reveals that real-world problems are more complex. In order to function efficiently, it is vital that the system avoid unnecessary work. In general, there are two reasons why modules present in a visual program may not need to be executed when their turn comes: 1) their results are not actually required by a result of the network and 2) their inputs are unchanged from the last time the module was executed (i.e. the result will be the same).

The outputs of a visualization network occur in modules that have side effects. They produce results outside of the visual program itself such as the display of images on a workstation or the creation of output files. Unless the result of a module ultimately affects a module that produces a side effect, that module does not need to be executed.

Eliminating modules that are not ancestors (i.e., not upstream) of modules with side effects is done in Data Explorer by preprocessing the network before the actual data-flow network evaluation commences. This is done by traversing the graph bottom-up, beginning at each module known to have side effects and flagging each module as it is encountered. Once this is complete, modules that have not been flagged do not have to be executed.

Note that the exact order in which modules will be executed cannot be controlled by the user; for example, modules in two parallel branches may execute in any order with respect to one another; it is only guaranteed that a module that depends on the results of one or more modules will wait for them to be complete before it is executed.

4.2 Iterative Execution and Caching of Intermediate Results

Unlike the simple example in Figure 18 on page 39 most real visualization problems involve some form of iteration. This may either be direct interaction, where the user is adjusting parameters of the visualization and observing their effect on the resulting images, or animation, in which one or more inputs to the network may vary from frame to frame.

In iterative applications, there are often major parts of the network that are unaffected when input parameters are modified. In Figure 18 on page 39 if the **isovalue** input to the Isosurface module is changed, only the affected module and its descendents need to be executed. The output of Import is not affected by the change. Hence, it can be reused, which avoids a superfluous access of data on disk. The MapToPlane module also does not need to be executed, since its inputs did not change.

One way to implement this capability is via a caching mechanism for partial results. Instead of immediately reexecuting when its inputs arrive, a module may first determine whether its inputs have changed. If they have not changed, it can simply retrieve its results from the cache. Otherwise, the module reexecutes, placing its new result into the cache.

Data Explorer extends this notion by incorporating a cache (implemented by the module scheduler rather than by the modules themselves) for *all* partial results. This cache retains results from not only the previous execution of the network, but from all prior executions (this is the default behavior; the user can also control cache settings for modules). The saving of objects in the cache is subject to memory limitations and a least-recently-used cache eviction strategy (items used the longest time ago are first to be discarded from the cache). The caching behavior for each output of a module may also be explicitly set by a user to optimize memory utilization. (See A.1, “Using Data Explorer Effectively” on page 212.)

The caching of partial results means that in general, the output of Import is held in the cache. Usually, this is highly desirable, as it avoids needing to reimport the data every time the visual program is run. However if you modify your file on disk (e.g., by editing it), Data Explorer will not know that the file has been changed, and will continue to use the cached version. To force Data Explorer to reimport the data, use the **Reset Server** option of the Connection menu. This will cause all items in the cache to be discarded, and Import will reaccess the file on disk. You may also set Import to cache *no results* by using the Cache option of Import’s Configuration dialog box; note, however, that this will not necessarily cause Import to run every time unless modules downstream from Import are also set to cache no results.

Note: An asynchronous module could be used to monitor a file’s status and generate new outputs when the file changes.

4.3 Conditional Execution: Route and Switch

Two of Data Explorer's mechanisms to control execution flow through a visual program are the Switch and Route modules. Switch allows you to switch between one or more inputs to drive a single output; Route is the inverse of Switch, having a single input that can be routed to zero, one, or more than one output. Switch is typically used to choose between different paths in a visualization program; for example, to pass an imported data field through either the Glyph module or through Isosurface, depending on user choice or characteristics of the data field itself. Route is typically used to turn off portions of the visualization program; for example, to turn off WriteImage or Export, or to prevent rendering to an image window unless the user chooses to create an image. Switch can be thought of as turning off portions of the visual program logically *above* Switch; Route can be thought of as turning off portions of the visual program logically *below* Route. Note that while Route turns off modules that receive its unused outputs, the Collect module is an exception: it runs even if some of its inputs have been turned off by Route.

Figure 19 shows an example of a Switch module controlling whether an Isosurface or a MapToPlane is passed to Image. In a simple data-flow execution model, this

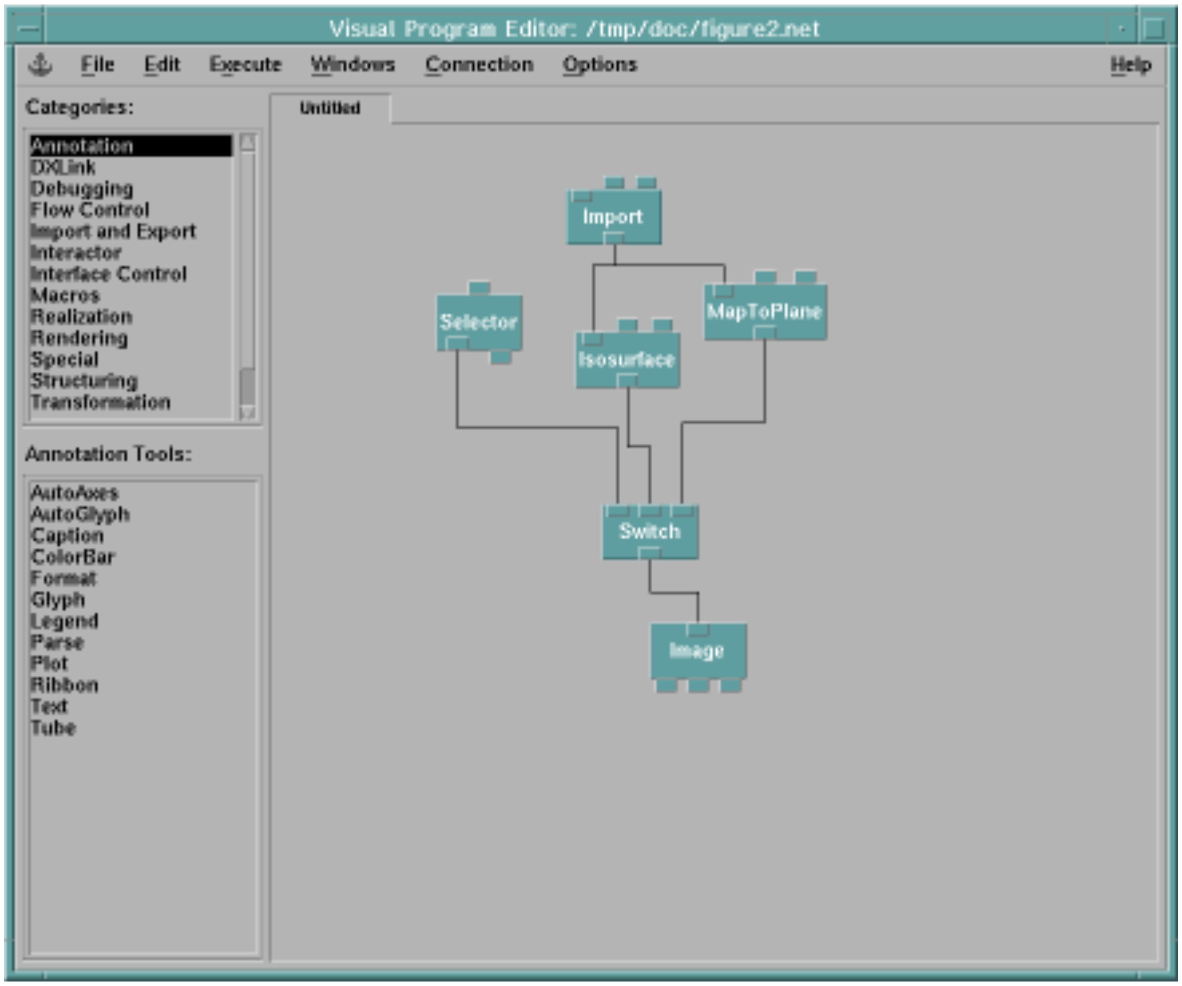


Figure 19. Example 2

Switch module will be executed when its inputs are available (i.e., the results of the Isosurface and MapToPlane modules, and the value of the selector). On execution, the Switch module chooses whether to pass the Isosurface or MapToPlane result to the output based on the **selection** input to Switch. In the case of a pure data-flow model both the Isosurface and MapToPlane modules execute before the decision as to which will actually be used is known. Since both operations can be computationally expensive, the execution of both of them is very inefficient.

Again, this problem is handled in Data Explorer within the simple data-flow execution module by an examination of the graph prior to execution. If the **selection** value comes from an external source (e.g., an interactor) and is known a priori, the selection may be performed by a simple transformation of the graph: excising the Switch module altogether, and substituting arcs from the selected source (either Isosurface or MapToPlane) to each of the modules that, in the original network, received the result of the Switch module. This leaves the unselected module dangling. It and any of its ancestors that are therefore made unnecessary will not be executed.

A different procedure is used if the controlling value is not static (e.g., if it is determined elsewhere in the network), as shown in Figure 20 on page 43. Suppose either an isosurface or a set of vector glyphs is selected depending on whether the data are scalar or vector. The determination of the type of the data is made using the Inquire module (i.e. at run time). In this case, the selection value for the Switch module cannot be determined before the execution of the graph. Instead, the graph must be evaluated in stages: 1) determine the selection value, 2) determine the necessary input to the Switch module and 3) evaluate the remainder of the graph. Since dynamic inputs may themselves be descended from other non-static inputs (e.g., computed in the network), this process may have to be performed repeatedly.

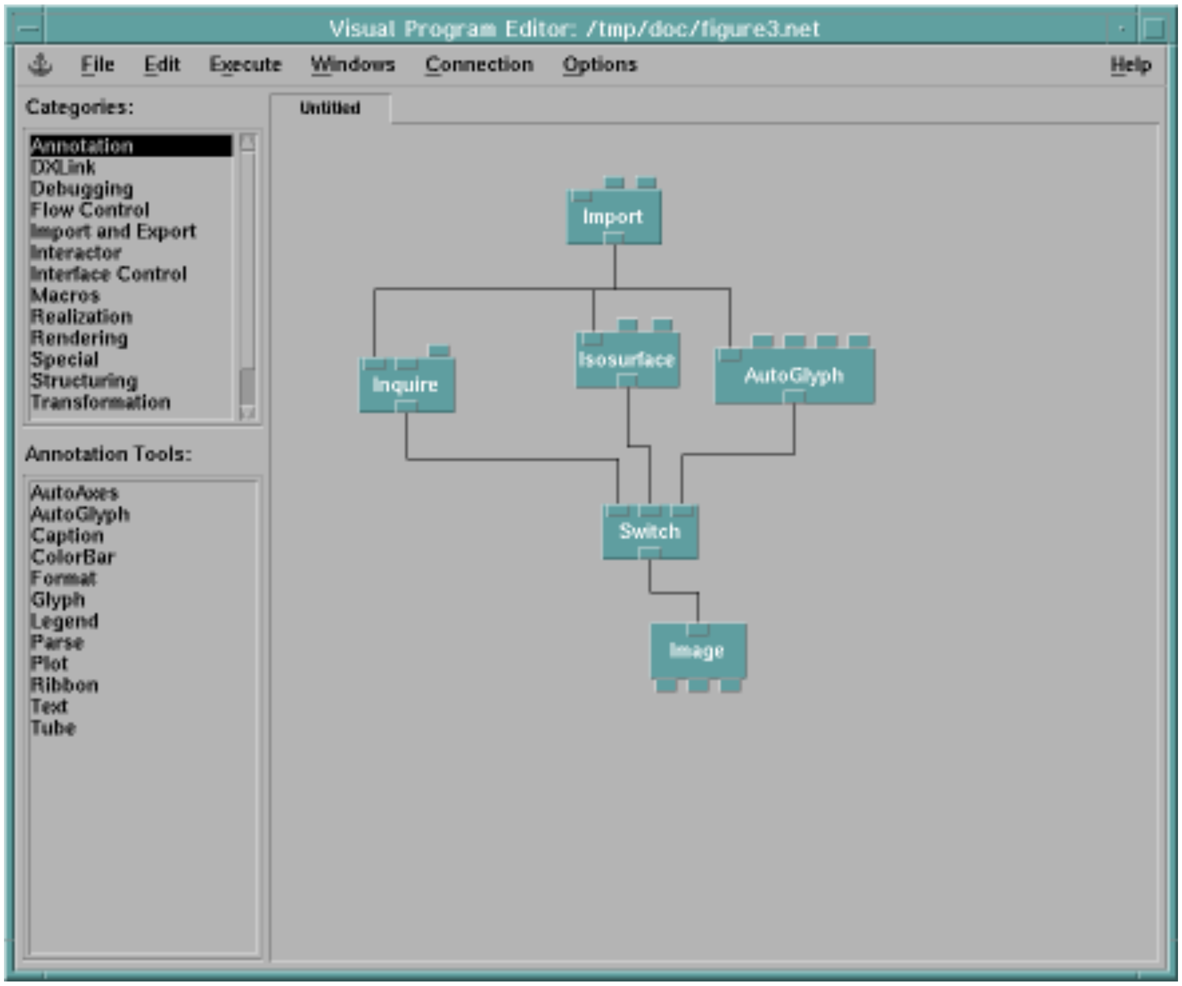


Figure 20. Example 3

4.4 Iteration using the Sequencer

Caching of intermediate results is particularly useful in conjunction with the Data Explorer Sequencer module. The Sequencer provides Data Explorer with a very simple and flexible animation capability. The sequencer outputs a frame count, which is updated with each new execution. The user controls the behavior using a VCR-like interactor (see “Using the Sequencer” on page 68).

The first time the Sequencer is “played”, it causes the network to be executed with each value for the Sequencer output. Each execution, which may be time consuming, will result in a new image being generated. These images are simply the result of a rendering module and will be retained in the cache. When the Sequencer is “replayed”, the inputs to the network are the same as they were for one of the frames in the first set of executions. Thus, the result of the execution (an image) will be immediately available from the cache. Hence, Data Explorer provides an automatic mechanism to create real-time animations even when the computation of each frame is slower than real-time.

The value produced by the Sequencer can be used in a number of ways. The Sequencer may be used to iterate through a time-dependent data set, causing the visualization to operate on each time step in turn, resulting in an animation showing how the data vary with time. As another example, the Sequencer could be used to drive the isovalue input to the Isosurface module.

4.5 Iteration using Looping

The sequencer provides a basic loop; however it has some limitations. Only one sequencer per visual program is allowed, and, as it executes, everything in the visual program executes (subject to the optimization for deciding which modules need to be executed, discussed above). Additional functionality is provided using the ForEachN, ForEachMember, Done, and First modules.

ForEachN and ForEachMember essentially implement a standard programming-language “for” loop; in the case of ForEachN iterating over a specified list of integers, and in the case of ForEachMember iterating over the members in a group or the items in a list (or array). Figure 21 shows a simple loop that outputs the integers 0 to 10 and Echo's them (the **`start`** and **`end`** parameters to ForEachN have been set to 0 and 10 respectively). This is roughly equivalent to the C-language statements

```
for (i=0; i<=10; i++)
   printf("%d", i);
```

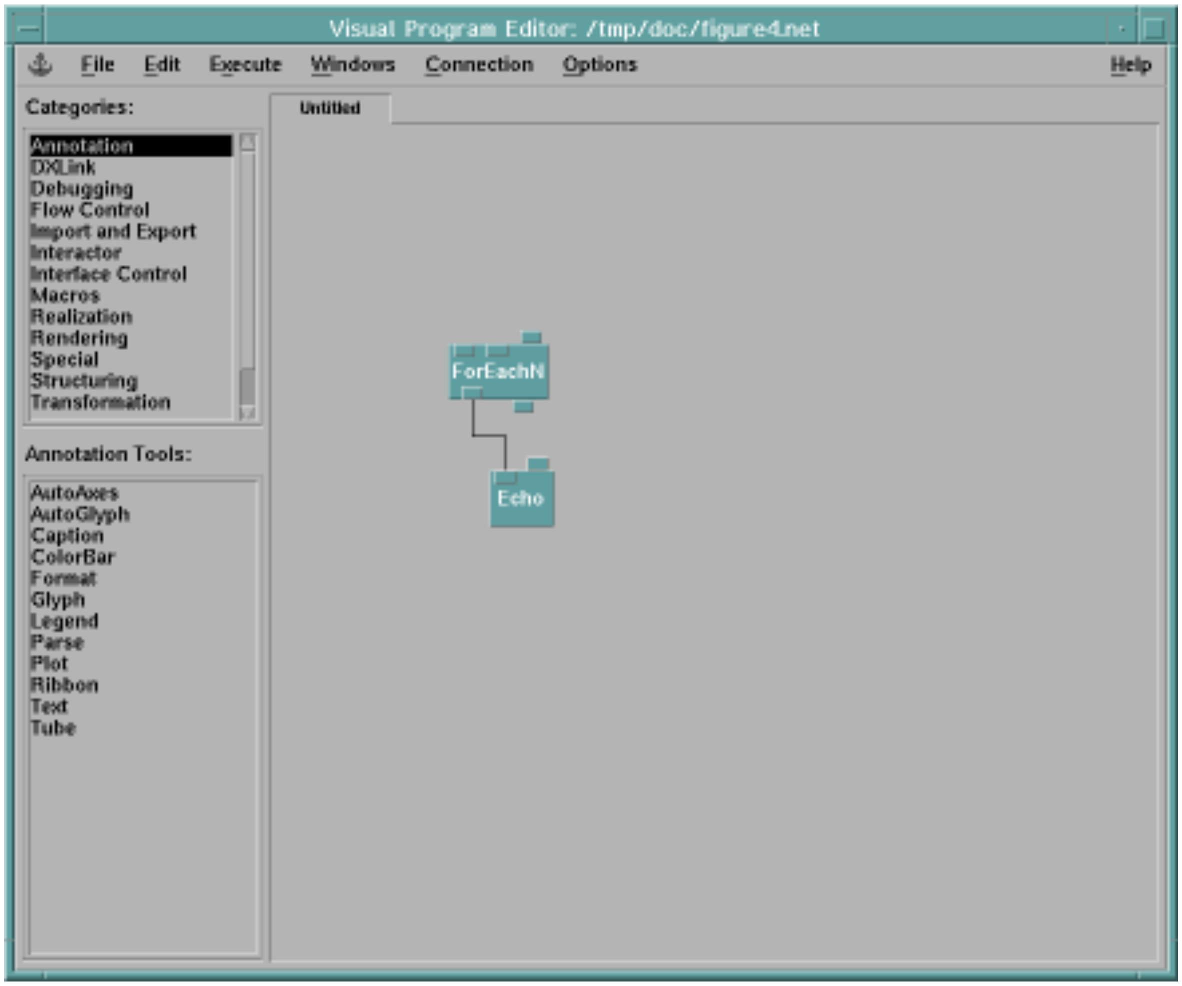


Figure 21. Example 4

Data Explorer provides you with two other tools for control looping, Done and First. Done enables you to exit a loop. Examples of its use can be found in “Advanced Loop Constructs” (see below). The First module provides a way to recognize the first pass through a loop; this is particularly useful, for example, as a way to reset the GetGlobal module if you are using GetGlobal and SetGlobal (see below) to store information during the execution of a loop. Note that First is not necessary if you are using GetLocal and SetLocal.

When a loop is present in a visual program, it causes the execution of all modules in the visual program containing the looping tool (ForEachN, ForEachMember, or Done), subject to optimization. For this reason it is strongly suggested that looping modules NOT be placed in the top level visual program, but rather be used only within macros. If used within a macro, the macro will not output any values until the loop is complete, when the ForEachN or ForEachMember list is exhausted or when the Done module causes an exit.

If a loop occurs inside a macro, and you reexecute a visual program calling this macro, the loop will not be reexecuted as long as the result of the macro remains in the cache. However, the presence of a side effect module (such as WriteImage or Print) inside of a loop will cause the loop to be reexecuted regardless of whether the output of the macro remains in the cache. If this is not the desired behavior, Route can be used to turn off the entire macro.

For efficiency you might find it desirable to set the caching option to **`Last Result`** for modules within the loop. In this way, multiple intermediate values within a loop will not use up valuable cache space.

Note that the full execution of a loop is considered to occur within a single execution of the graph (as would occur if you select Execute Once from the Execute menu). Thus if you change any interactor values DURING the execution of the loop, those interactor values will not take effect until the loop is complete. This is an important way in which looping differs from using the Sequencer; if you change the value of an interactor while the Sequencer is running, the value will be updated on the next frame of the sequence.

4.6 Preserving Explicit State

Some visualization applications require the retention of state from one execution to the next, which as discussed earlier, cannot be supported within the context of pure data flow. Consider, for example, the creation of a plot of data values at a given point while sequencing through a time series. The state of the plot from the prior execution is retrieved. It is updated by appending the new time-step information, and the result is then preserved by resaving the state of the plot for the next execution. Data Explorer provides two sets of tools for preserving state depending on whether the state needs to be preserved over one execution of the network or over multiple executions of the network. The tools for preserving state are GetLocal, SetLocal, GetGlobal, and SetGlobal. The Set tools enable you to save an object (in Data Explorer's cache) for access in a subsequent execution or iteration. The Get tools enable you to retrieve the object saved by the Set tools.

You pair a GetLocal and SetLocal in a visual program by creating an arc from GetLocal’s **`link`** output parameter to SetLocal’s **`link`** input parameter. In a visual program a GetLocal typically appears logically above a SetLocal. When GetLocal runs, it checks if an object has been saved in the cache. If no object was saved

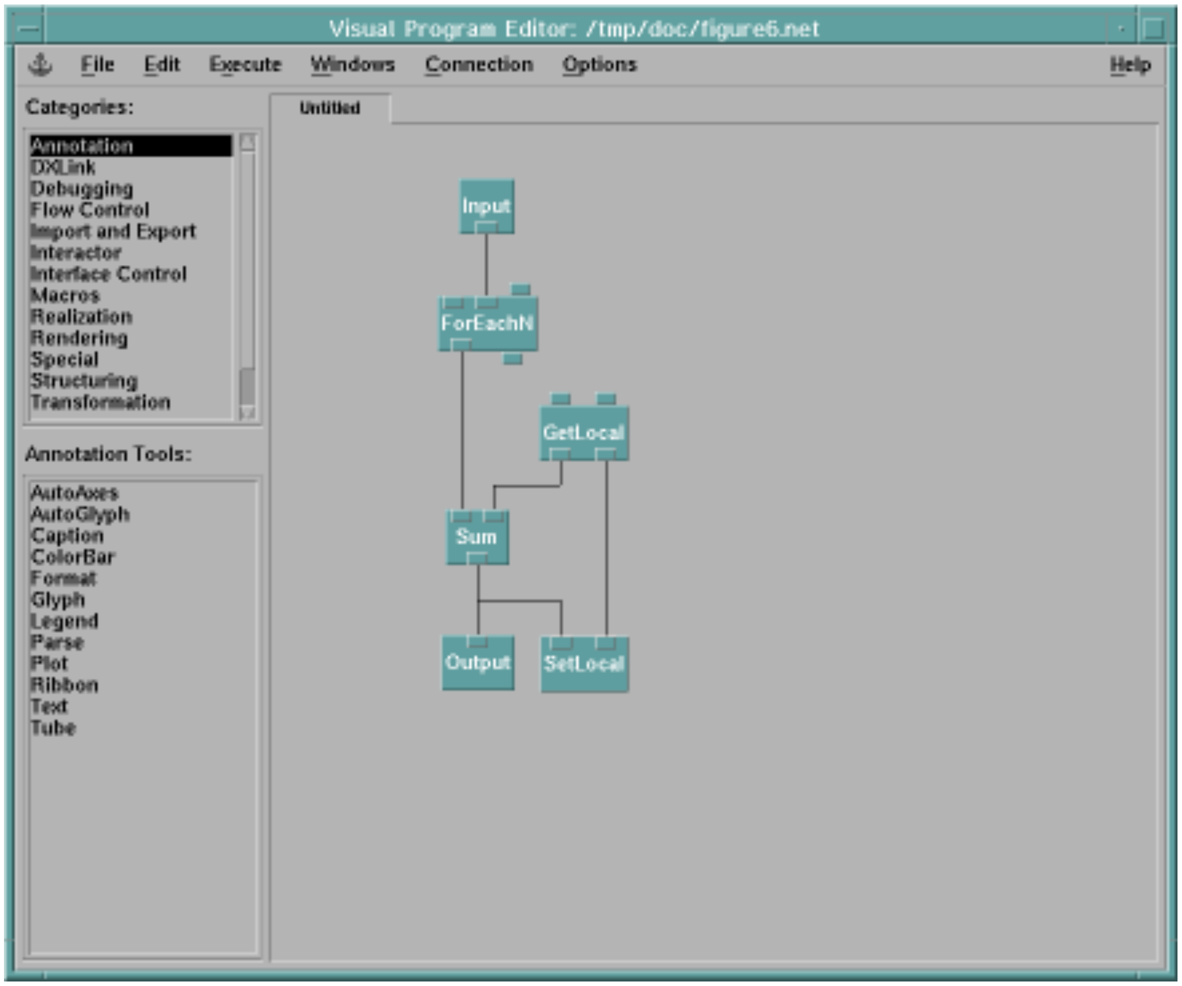


Figure 22. Example 5

(as would be the case if SetLocal has not yet run) or the **reset** parameter to GetLocal is set, GetLocal outputs an initial value that you can set using the **initial** parameter. Otherwise, GetLocal retrieves the saved object from the cache and outputs it. When SetLocal runs, it saves its input object in the cache and then indicates that its paired GetLocal should simply be scheduled during the next iteration of a loop or the next time an execution is called for. (Note that if GetLocal is inside a macro, it will be executed only if the macro needs to be executed; that is, if the macro's inputs have changed or there is a side effect module in the macro.)

GetGlobal and SetGlobal are paired in the same way as GetLocal and SetLocal. They also save and retrieve items from the cache. The main difference is that GetGlobal and SetGlobal will preserve state over more than one execution of a program. (However, recall that a complete loop takes place within a *single* execution.)

Using GetGlobal and SetGlobal is comparable to using a static variable in C-language programming. GetLocal and SetLocal are good for saving state inside of a looping construct. Once the loop is terminated, the state is reset for the next execution of the loop. To save state in a program that uses a Sequencer module, you should use GetGlobal and SetGlobal, since each iteration of the Sequencer is a separate execution of the program as described in 4.5, “Iteration using Looping” on page 44.

Illustrated in Figure 22 on page 46 is a simple macro that sums the numbers from 1 to N, where N is an input parameter. The **start** parameter to ForEachN has been set to 1. GetLocal and SetLocal are used to accumulate the sum. Sum is a trivial macro consisting of a Compute where the expression is “a+b.” On the first iteration of the loop, GetLocal will output its **initial** value, which has been set to 0. On subsequent iterations GetLocal will output the accumulated value SetLocal saved during the previous iteration. When the loop terminates the final accumulated value is the output of the macro. This macro is roughly equivalent to the following C-language statements:

```
b = 0;
for (a=1; a<=10; a++)
  b = b+a;
```

If the macro were run again, on the first iteration of the loop GetLocal would again output its **initial** value. (Note that the macro will only run again if the input to the macro changes or the output of the macro has been removed from cache.)

If you replaced the GetLocal and SetLocal in Figure 22 on page 46 with GetGlobal and SetGlobal it would be equivalent to the following C-language statements:

```
int a;
static int b = 0;
for (a=1; a<=10; a++)
  b = b+a;
```

While when SetLocal is used, the sum is reset each time the macro is run, if SetGlobal is used, the sum of a previous execution is added to the sum of the current execution. For example, let macro_local be the macro shown in Figure 22 on page 46 and macro_global be the same macro but with SetGlobal and GetGlobal substituted for SetLocal and GetLocal. If the input to both macros is 10 then both macros will output 55 (the sum of numbers 1 to 10) the first time they are run. If an execution takes place without the input to the macros changing then neither macro will run again and the value 55 will be used as the output again. If you change the input to 3 then macro_local will output 6 and macro_global will output 61 (55+6).

Illustrated in Figure 23 on page 48 is a macro that returns the accumulated volumes of the members of a group and the number of members in the group. ForEachMember is used to iterate through the group. Measure is used to determine the volume of a member and the GetLocal and SetLocal pair on the left side of the macro is used to accumulate the volumes. For illustrative purposes, a loop containing GetLocal, SetLocal, and Increment is used to count the number of members in the group. (Inquire also provides this function, as does the **index** output of ForEachMember.) Increment is a trivial macro consisting of a Compute where the expression is set to “a+1.” The **initial** values to both GetLocal tools are 0.

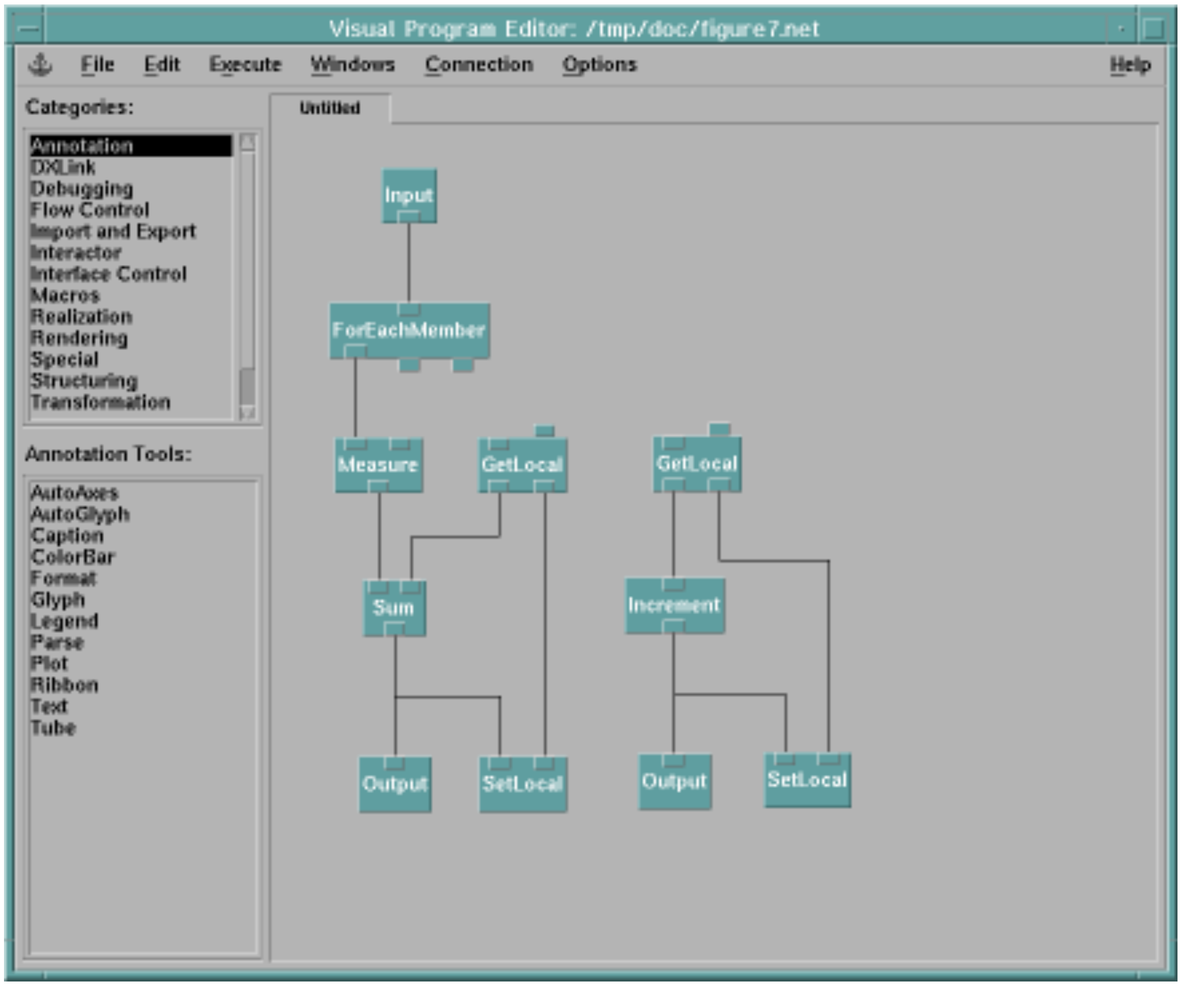


Figure 23. Example 6

Illustrated in Figure 24 on page 49 is a visual program that saves the current camera settings for use in the next execution of the program. The initial value of GetGlobal is NULL. The Inquire module checks to see that the output of GetGlobal is a valid camera object. If it's not a camera object, then Route is used to ensure that the Display module is not scheduled to run. When a new camera is chosen (for example by rotating the object in the Image window) the Display window will show the image using the previous execution's camera settings.

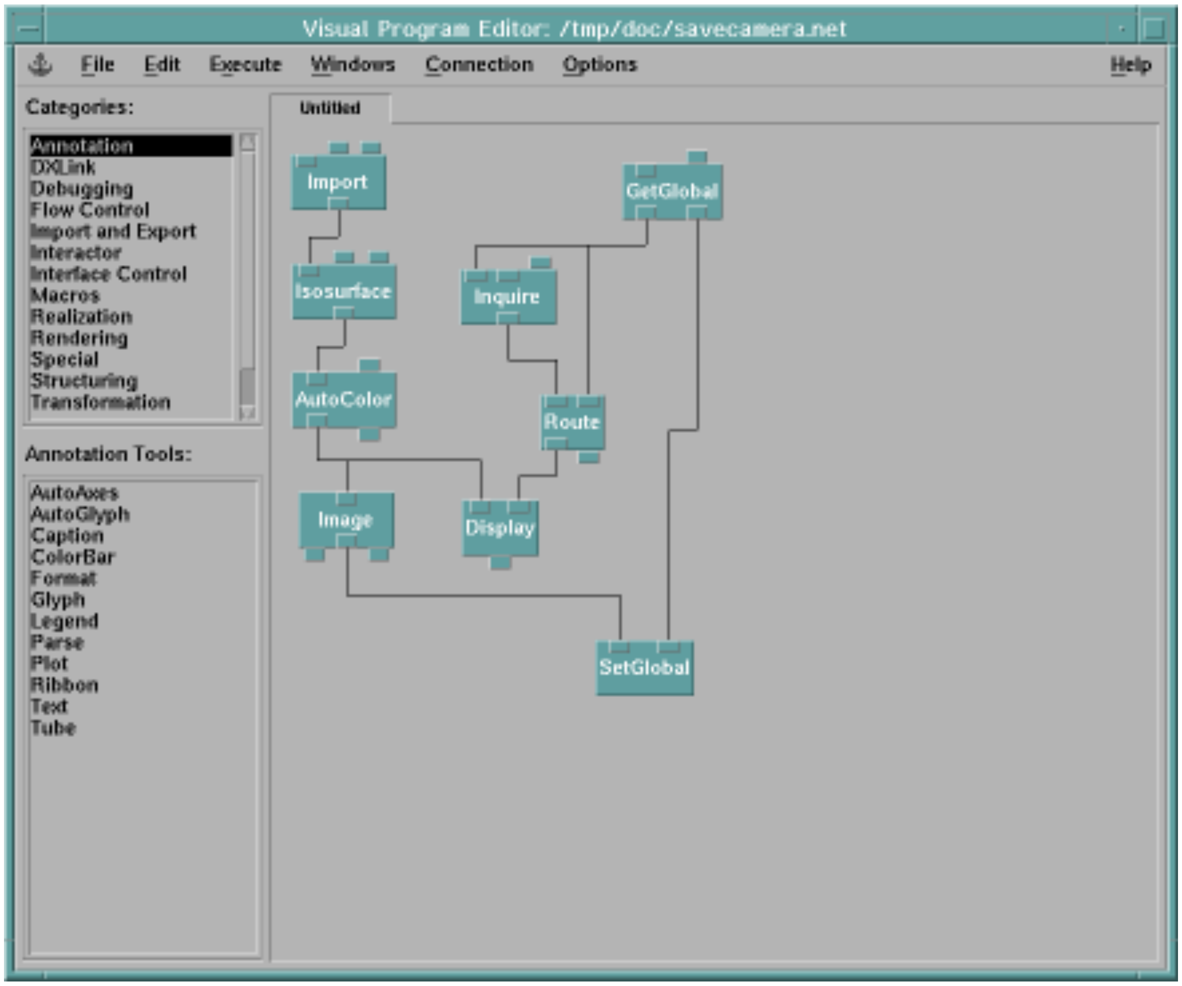


Figure 24. Example 7

As mentioned previously, in a true data-flow implementation, all modules are pure functions (i.e. their outputs are fully defined by their inputs). Hence, processes are stateless with no side effects. A macro in Data Explorer is considered to be a function, with its outputs being fully defined by its inputs. This is no longer true when a GetGlobal module is added to a macro. GetLocal maintains state information only within one execution of the macro. GetGlobal maintains state information between executions, and therefore the outputs of a macro containing GetGlobal are no longer entirely defined by the inputs. The outputs from macros with state (containing a GetGlobal module) are guaranteed to stay in the cache until the inputs for that macro change. At that point, the results of the previous execution are discarded to make room for the new results. This is equivalent to setting the cache attribute of the macro to `cache last` for each of the outputs. These cache settings cannot be overwritten by the user. This guarantees coherency when executing macros with state.

4.7 Advanced Looping Constructs

Combinations of the modules described above enable you to create advanced looping constructs. These constructs are equivalent to C-language constructs such as “do while” or “for” loops containing “break” and “continue” statements. In the following figures the Sum and Increment macros, as described above, are used as well as a macro named Equals that consists of a Compute where the expression is “a==b?1:0” (if the inputs are equal output 1 otherwise output 0).

Illustrated in Figure 25 on page 51 is a macro that computes the sum of numbers from 1 to N. If a number in the sequence from 1 to N is equal to an external input, x, the loop terminates and returns the sum from 1 to x. Done, in combination with Equals, is used to cause early termination of the loop. Done causes the loop to terminate after all the modules in the macro have executed if the input to Done is nonzero. The macro illustrated in Figure 25 on page 51 is equivalent to the C-language statements:

```
sum = 0;
i = 0;
do
{
   i++;
   sum = sum+i;
} while (i<=n && i!=x);
```

Now consider a macro in which the sum of numbers from 1 to N is computed, but if a number is equal to an external input value, x, it is excluded from the sum. To achieve this result using C-language statements, you would use a conditional with a “continue” statement:

```
sum = 0;
for (i=1; i<=n; i++)
{
   if (i==x) continue;
   sum = sum+i;
}
```

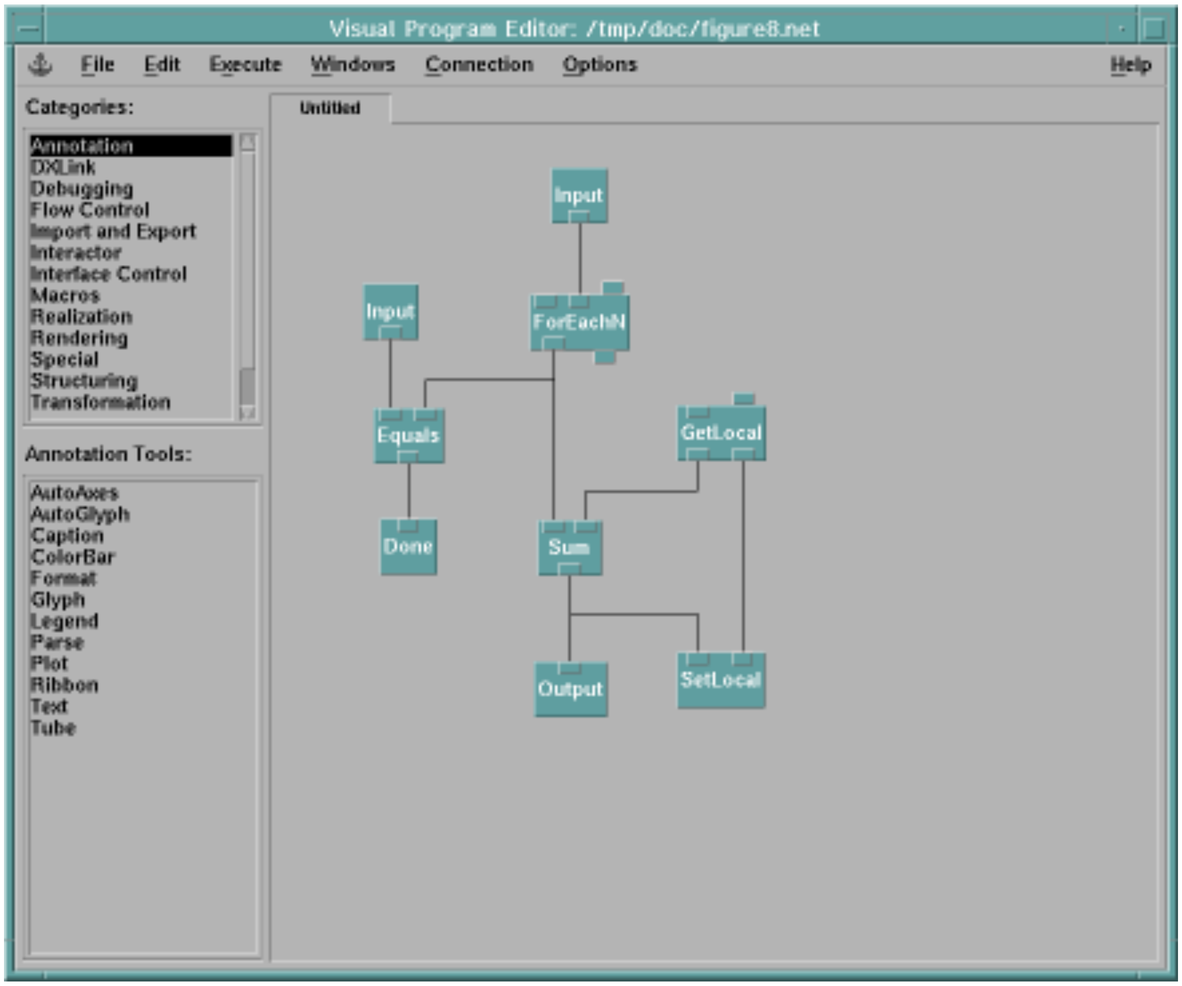


Figure 25. Example 8

As illustrated in Figure 26 on page 52, you would use Route to create this macro using Data Explorer. The **`selector`** input of Route is being controlled by the output of Compute. The Compute has its expression set to “a==b?0:1” (if a and b are equal output 0, otherwise output 1). (This is similar to the Equal macro used earlier, but the expression differs slightly.) Therefore, if the iteration variable is equal to x, Compute outputs a 0, causing Route to disable the execution of all the modules downstream from it. This implies that Sum and SetLocal will not run; therefore, during the next iteration, GetLocal will retrieve the same value as the current iteration.

Unfortunately, the visual program illustrated in Figure 26 on page 52 has a minor problem. If x equals N, the Route will cause the Sum and SetLocal not to execute during the last iteration; therefore the output of the macro will be a NULL.

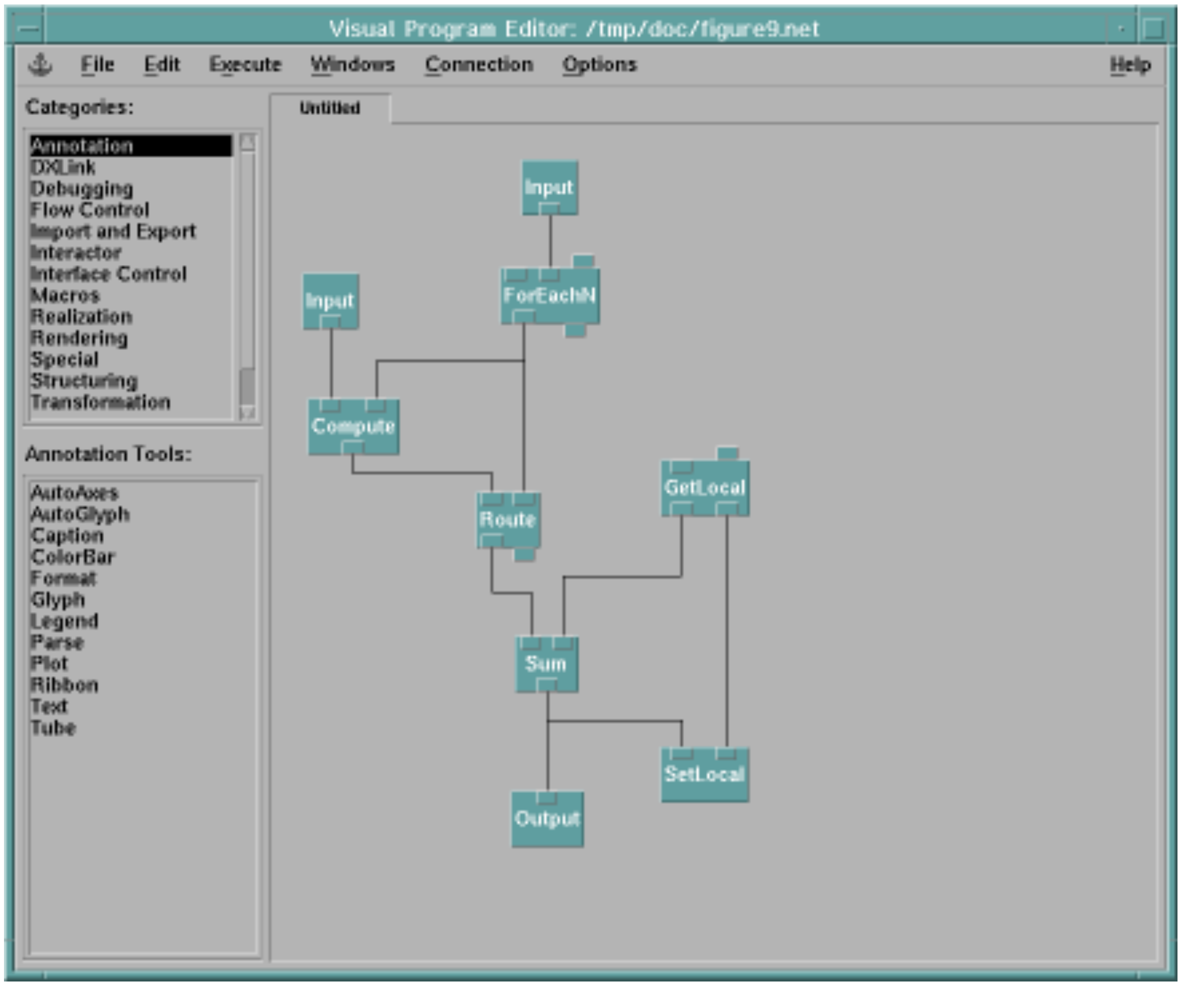


Figure 26. Example 9

Illustrated in Figure 27 on page 53 is the fix to the problem. A Switch is included to choose the correct input for the output of the macro. If x equals N, the output of the GetLocal is chosen; otherwise the output of Sum is chosen.

If you want to create a loop containing an early exit in the middle of the loop (a “break”), you need to use a Route in combination with Done. Illustrated in Figure 28 on page 54 is a macro that performs the equivalent function as the C-language statements:

```
sum = 0;
for (i=1; i<=n; i++)
{
   if (i==x) break;
   sum = sum+i;
}
```

Data Explorer allows you to have multiple Done tools in a single loop enabling you to have more than one break or continue or combinations of the two.

ForEachN or ForEachMember simplify the use of loops but they are not necessary for creating them. In fact, Done itself is sufficient, if it is included inside a macro. The macro will execute repeatedly as long as the **done** parameter is equal to 0

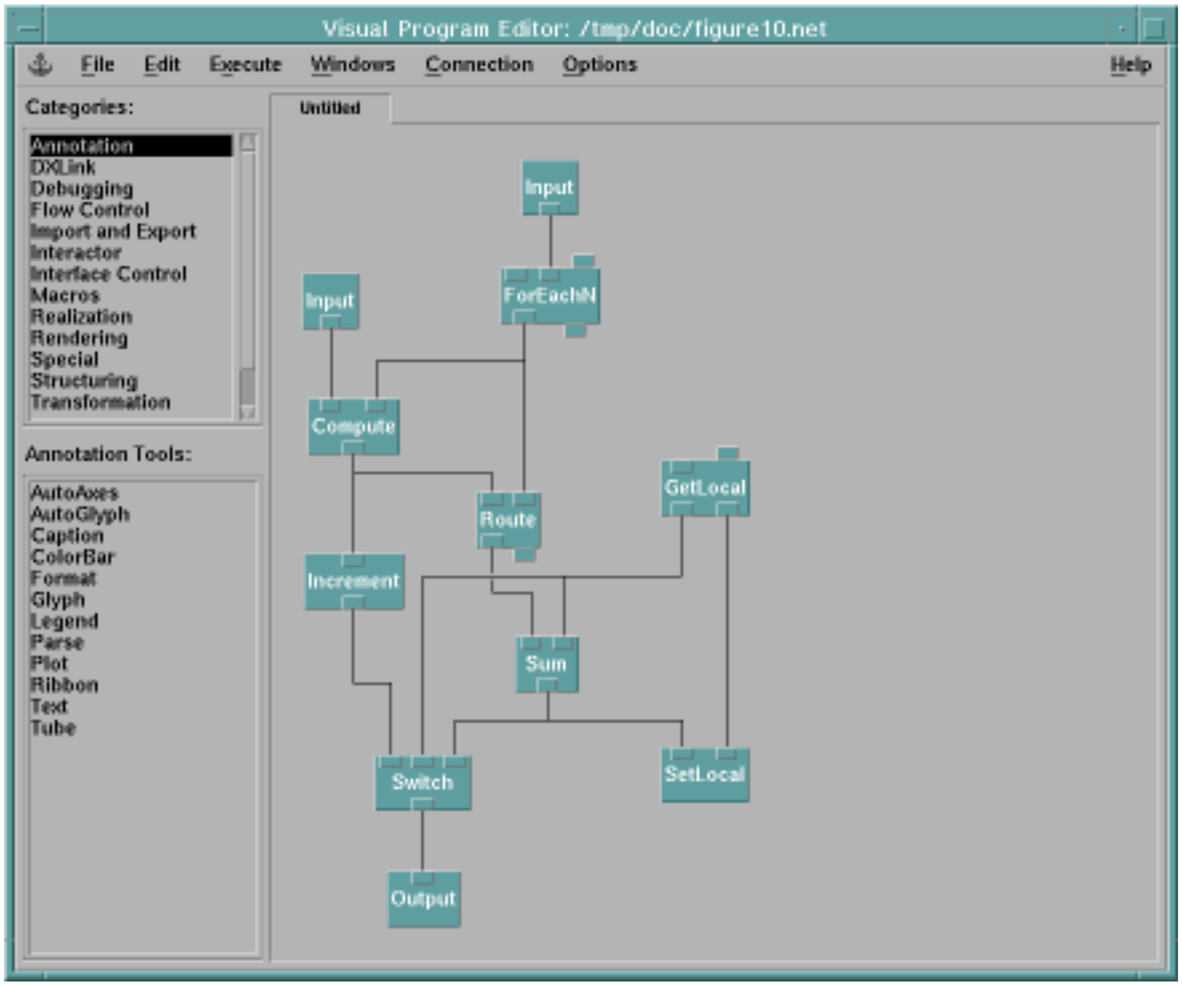


Figure 27. Example 10

(zero). Note that the top-level visual program is itself a macro, so the same behavior will occur if Done is placed in the top-level visual program.

Illustrated in Figure 29 on page 55 is a macro that computes the Fibonacci Series (defined by setting Y_1= 1, Y_2 = 1 and by the recursion formula $Y_k = Y_{k-2} + Y_{k-1}$, for k = 3,4,5...). In this example a two vector, [Y_{k-1}, Y_k], is used to store the elements of the series. The GetLocal module has its initial value set to [1,1]. The first Compute in the macro creates a new two vector consisting of [Y_{k-1}, Y_k] using the expression "[a.1, a.0 + a.1]." The second Compute in the macro extracts Y_k from the two vector using the expression "a.1." To terminate the loop, the Y_k element of the series is checked against an external input, x. If Y_k is greater than x, the loop terminates. GreaterThan is a simple macro consisting of a Compute with its expression set to "a>b?1:0." An equivalent set of C-language statements is:

```
a=1;
b=1;
do {
   c = b;
   b = b + a;
   a = c;
} while (b <= x);
```

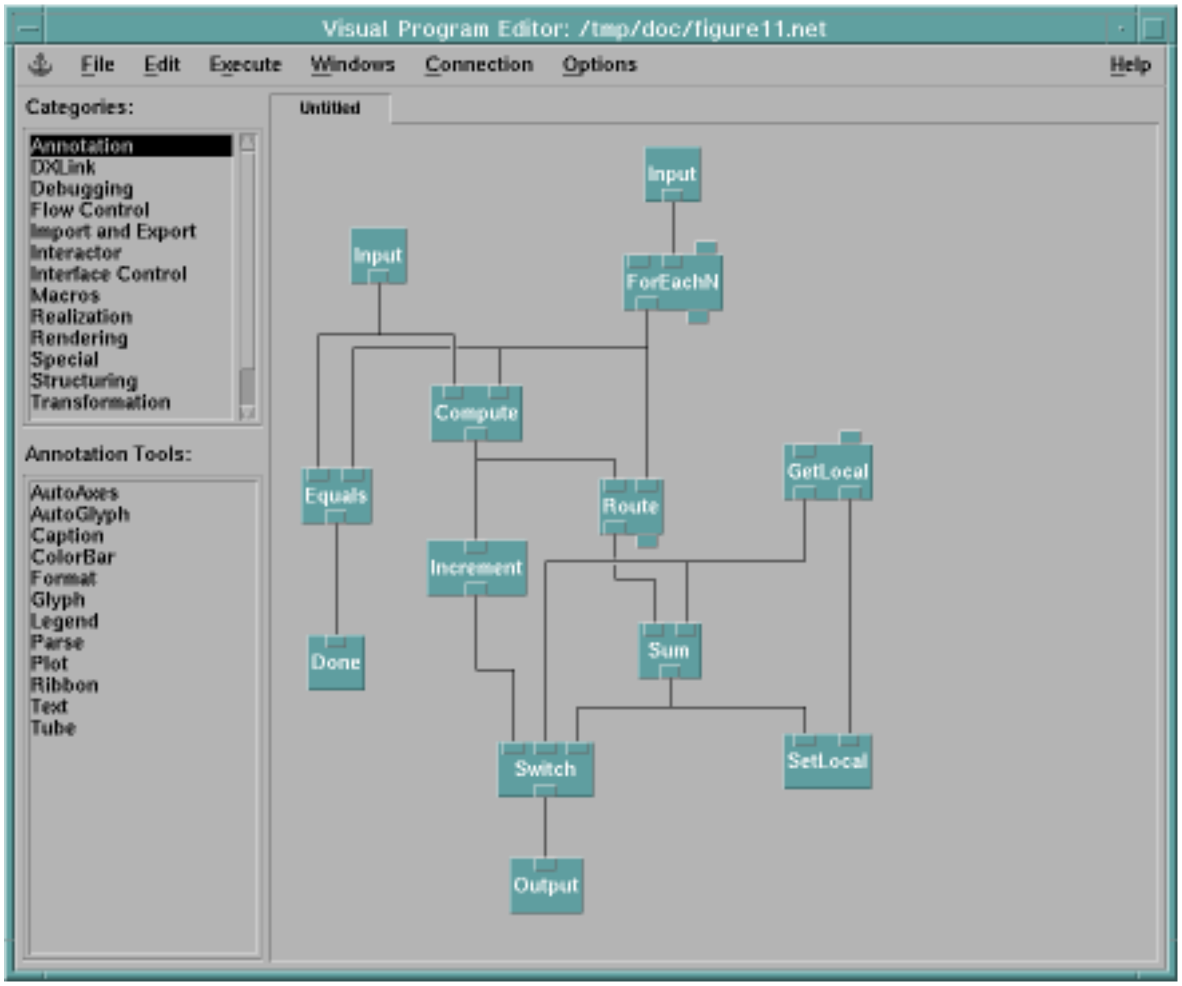


Figure 28. Example 11

4.8 External Asynchronous Data Sources

Many applications of visualization tools call for a direct interface to external data sources, especially ones that generate data to be studied (e.g. a computational simulation). The execution model of Data Explorer provides the framework for real-time visualization of data generated asynchronously by such a process. An external data source is linked into a Data Explorer network by incorporating a communications module, which receives data from the external source, often across a socket, and passes the resulting data object to the module's output. This module (and its descendents) will only run when the external data source has indicated that new data are available (see 10.2, “Asynchronous Modules” on page 84 in *IBM Visualization Data Explorer Programmer’s Reference*).

Data Explorer also provides a mechanism for direct manipulation of the executive (e.g., mode, passing data, error handling, etc.) and the user interface (e.g., window visibility and mode) from an external application. This allows control of Data Explorer from other software and peer-to-peer communications (see Chapter 16, “DXLink Developer's Toolkit” on page 157 in *IBM Visualization Data Explorer Programmer’s Reference*).

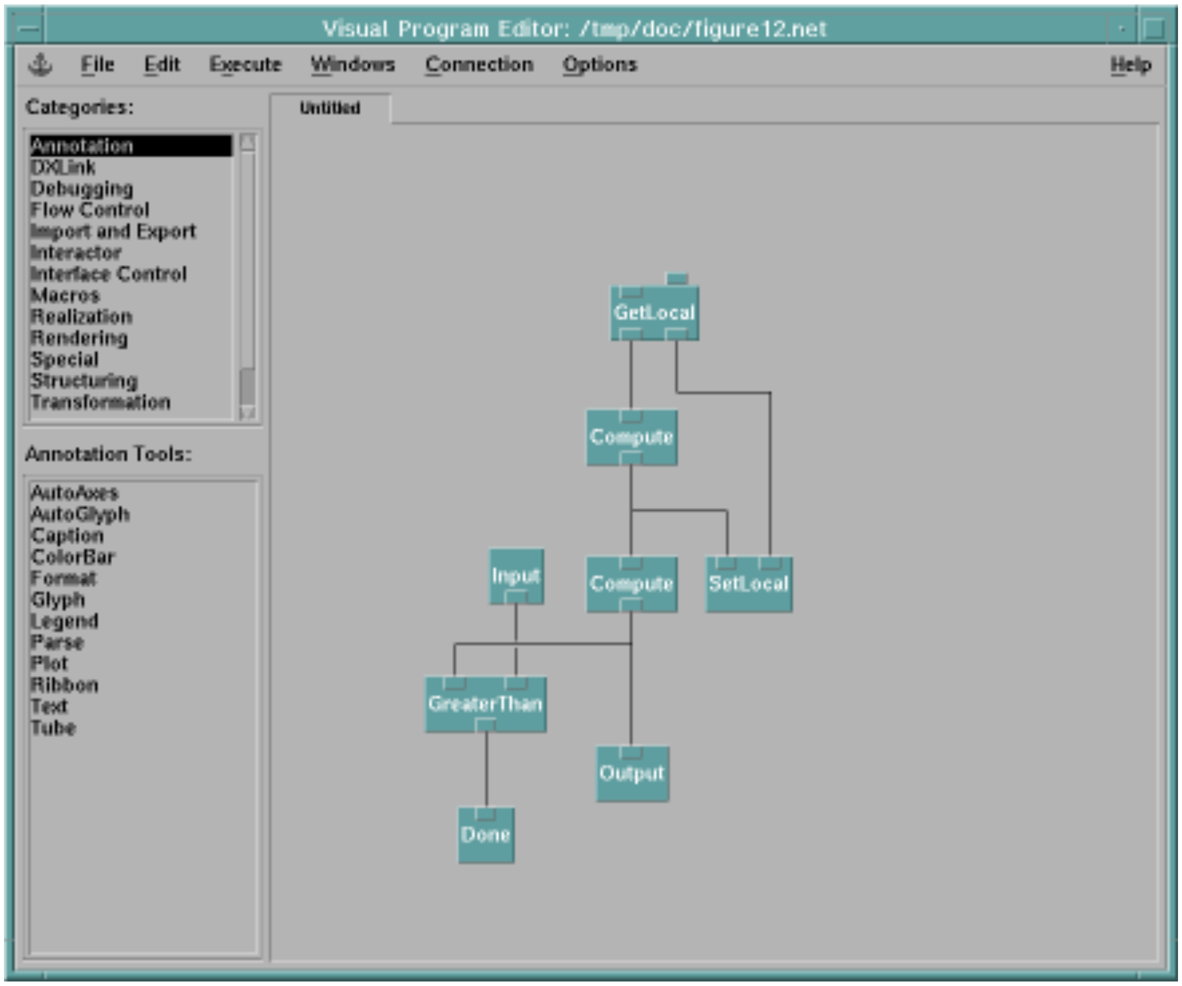


Figure 29. Example 12

4.9 Parallelism using Distributed Processing

Data Explorer provides the capability of distributing the execution of a visual program or a program generated using the scripting language over multiple workstations on a network. Distributing the execution provides parallelism and enhanced resource utilization. Parallelism is achieved by the simultaneous execution of different portions of the visualization on each of the workstations. Enhanced resource utilization can be achieved, for example, by assigning computationally intensive portions of the visualization to the more powerful workstations, or transformation and realization functions that are applied to data located remotely can be distributed to the remote workstations, reducing the amount of data transfer.

Distributed processing is achieved in two ways: using “outboard” modules or placing groups of tools into “execution groups.” These two methods can be used independently or in combination. An outboard module is a user-written module controlled by the Data Explorer executive but is external to the Data Explorer server program. They can be invoked from either a visual program or a script program. Execution groups are a set of tools that can be assigned to a workstation. Once groups are created, they can be assigned to the workstations over which the visualization is to be distributed. More than one group can be assigned to each workstation. (See also 9.1, “Using Distributed Computation” on page 178.)

4.10 Parallelism for Data Explorer SMP

For completeness, the notion of module parallelism is discussed here. If you are developing visualizations or modules exclusively for use with the IBM Visualization Data Explorer running on a single-processor workstation, then these concepts are not applicable. However, if your visualizations or modules are to be run on both the IBM Visualization Data Explorer and IBM Visualization Data Explorer SMP, then these concepts are important for achieving higher performance.

Every module that performs any significant amount of processing is “parallelized”; that is, the module makes use of all processors made available to Data Explorer to operate on the data.

Data Explorer uses explicit data partitioning as the primary framework for parallelism. Data Explorer partitions the data into local, self-contained regions. In general, visualization modules then generate subtasks corresponding to partitions. For more information about partitioning, see Partition in *IBM Visualization Data Explorer User’s Reference*.

In general, parallel programming is complex. To help manage it, Data Explorer simplifies the process by providing a simple fork-join parallelism model to implement coarse-grain shared memory parallelization (data parallel). Using data partitions, read-only objects, and a single-fork join mode simplifies the module writing task by avoiding the explicit use of locks in modules, thereby reducing the possibility of deadlock. For information about adding modules to the Data Explorer system, see *IBM Visualization Data Explorer Programmer’s Reference*.

Chapter 5. Graphical User Interface: Basics

This chapter describes how to use the graphical user interface provided with Data Explorer. This interface enables you to create and control the visualization of data. An image is created by applying data, as input, to a visual program. The visual program is a sequence of interconnected functions acting on one or more inputs and producing one or more outputs. Typically, input is user data and output is a realization of that data in the form of an image.

Data Explorer consists of the following windows:

Startup Window
: You can use the Startup window to access other windows in Data Explorer.

Visual Program Editor
: You can use the Visual Program Editor (VPE) to create and alter visual programs and macros.

Control Panel
: You can use Control Panel windows to set and control the input parameters of visual program tools.

Image
: You can use the Image window to view an image created by Data Explorer.

Colormap Editor
: You can use the Colormap Editor to map colors and opacity to specified data.

Sequencer
: You can use the Sequencer to advance through a data series or to step through a changing sequence of input parameters.

Message Window
: You can use the Message window to access error and working information about your execution.

Help
: You can use the Help window to access online documentation.

Data Prompter
: A graphical user interface that simplifies the import of data.

Module Builder
: A graphical user interface that simplifies the process of adding modules to Data Explorer.

For information on the Data Prompter, see *IBM Visualization Data Explorer QuickStart Guide*; on the Module Builder, *IBM Visualization Data Explorer Programmer's Reference*.

5.1 Starting Data Explorer

To run Data Explorer using a workstation, you must have:

- An account on the workstation
- If the executive portion of Data Explorer is to be executed on a remote workstation, then a `.rhosts` file in your home directory should contain the name of the machine on which the user interface will run. The permissions for the `.rhosts` file should be set to 600 (read and write only by the owner).

You can start Data Explorer in any of several modes. In all cases, you must first do the following:

1. Log on to your workstation.
2. Start an X Window System running with the Motif or the appropriate window manager.

If you are going to create, modify, and execute visual programs, start Data Explorer initially in the Visual Program Editor and connect to the executive (server) by typing:

```
dx -edit [program]
```

where *program* (which is optional) names an existing visual program.

If you plan to execute a previously created visual program, start Data Explorer initially in the Image window and connect to the executive (server) by typing:

```
dx -image program
```

or

```
dx -menubar program
```

where *program* is the name of an existing visual program.

You may want to start the user interface without connecting to the executive. For example, you may want to work on a visual program or macro, but may not plan to execute it until some later time. In that case, you can start Data Explorer with the following command:

```
dx -edit -uionly
```

Once the user interface is started, you can connect to the executive portion of Data Explorer by following the procedures described in 9.3, “Connecting to the Server” on page 183.

All of the command line options for Data Explorer are described in C.2, “Command Line Options” on page 295.

Using Environment Variables

There are several environment variables that you may find useful to customize Data Explorer. These can be set in your login profile, or set as required.

DXMACROS: The DXMACROS environment variable is a list of the directories in which Data Explorer will look for macros. If you do not specify DXMACROS, you will need to load macros individually, using the process described in 7.2, “Creating and Using Macros” on page 149. The directories are searched in the order in which they are specified in the environment variable. If multiple macros with the same name are encountered, the first macro found is used.

An example of the statement setting the DXMACROS environment variable (in the C shell environment) is the following:

```
setenv DXMACROS /usr/mydirectory/projectAmacros:/usr/mydirectory/projectBmacros
```

where **/usr/mydirectory/projectAmacros** and **/usr/mydirectory/projectBmacros** are two directories in which macros will be sought. Multiple directories can be listed, with directory names separated by a colon.

DXDATA: The DXDATA environment variable specifies a list of directories in which Data Explorer will search for data files. If the data you wish to import are in one of the directories specified in the DXDATA environment variable, then you do not need to specify the complete path name to the data in the Configuration dialog box for the Import tool. You can simply specify the file name, and the Import module will look in the specified directories for the data file. The directories are searched in the order in which they are listed in the environment variable; and the first occurrence of the data file is used.

An example of a statement that sets the DXDATA environment variable (in the C shell environment) is the following:

```
setenv DXDATA /usr/mydirectory/mydata:/usr/group/groupdata
```

where **`/usr/mydirectory/mydata`** and **`/usr/group/groupdata`** are two directories that contain data files. Multiple directories can be listed, with directory names separated by a colon.

DXHOST: The DXHOST environment variable is the initial machine name of the workstation on which to run the executive. If DXHOST is not specified, then a default of “localhost” is used. See 9.3, “Connecting to the Server” on page 183 for more information on how to connect to the server. The host name should be the name that results when you issue the **`uname -n`** shell command.

The rest of the environment variables and start-up options are discussed in Appendix C, “Environment Variables and Command Line Options” on page 291.

5.2 Understanding Data Explorer Windows

The Data Explorer user interface is built on the X Window System and Motif standards. These tools manage the windows used with Data Explorer. The windows you use depends upon how you choose to use the system.

The primary windows are listed and described at the start of this chapter. The primary window in which you begin a Data Explorer session (either the Visual Program Editor or Image window) is called the *anchor window*. (You can also have a menu bar as your anchor window by specifying `-menubar` on the command line.) This is the window that, if closed, ends the Data Explorer session. The anchor window is identified by a symbol resembling a ship’s anchor located in the top left corner, as illustrated in Figure 30 on page 61.

Descriptions of the primary window pull-down menu options are located in Section 8.1, “Using the Primary Window Pull-Down Menus and Options” on page 156.

Secondary windows, such as dialog boxes or informational boxes, appear when they are needed to complete a task. You can move secondary windows, but you cannot put them behind the primary window from which they came. See *IBM AIXwindows User’s Guide* or the appropriate window system overview for more information on how to manipulate windows.

In this chapter, when references are made to the X Window System it means any window server that supports X11 protocol, including Sun’s OpenWindows. The Motif window manger, mwm, has been used in many figures and examples in this chapter. Please use the appropriate window manager for your system, such as vuewm for Hewlett-Packard, 4dwm for SGI, and olwm for Sun. Since title bars and window borders are features of a window manager, the appearance of your windows may differ slightly from those in the figures and examples.

Looking at Window Structure

Figure 30 points out the major parts of a Data Explorer window. Definitions of the parts follow the figure.

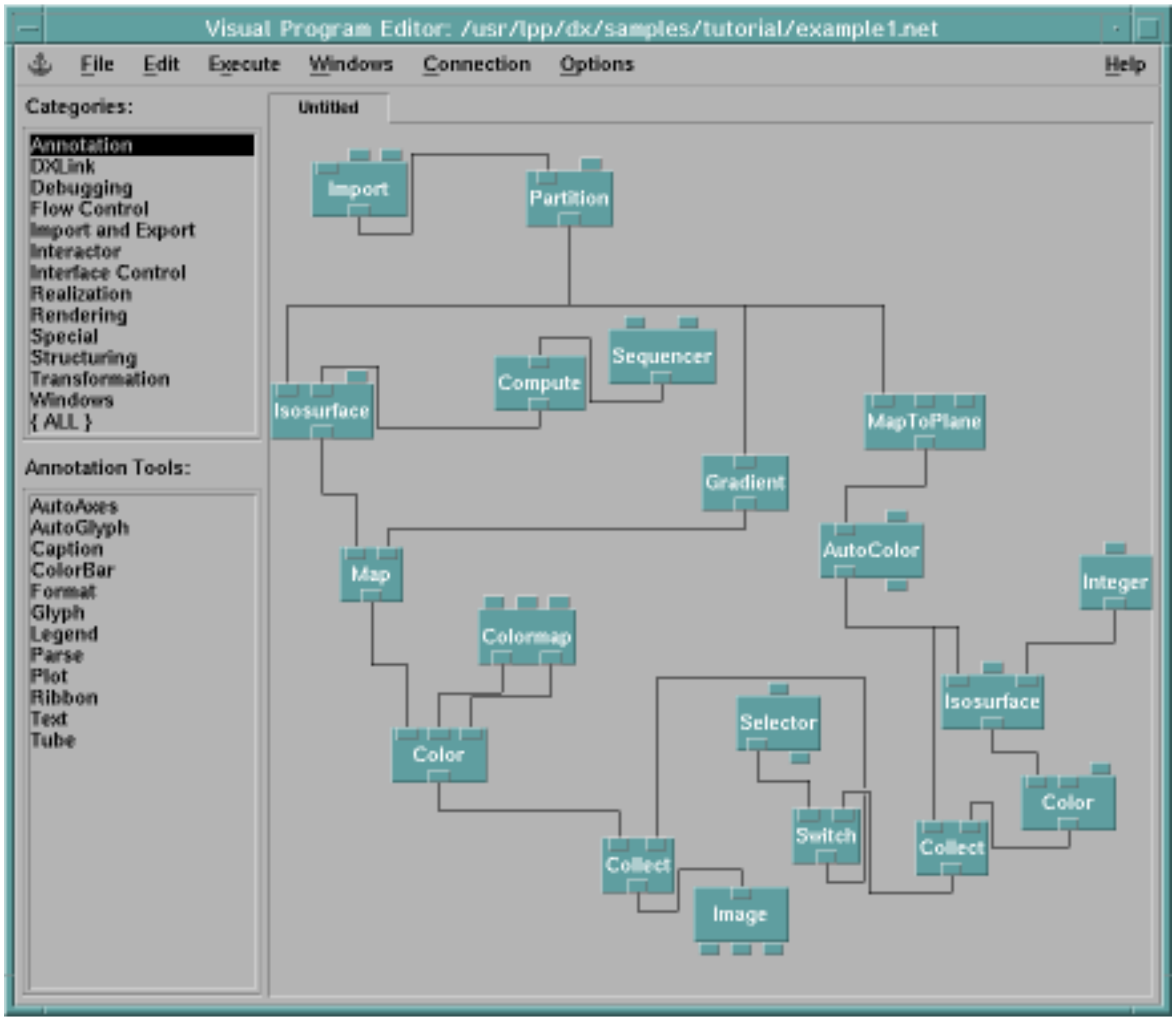


Figure 30. Visual Program Editor Window

The following are features of Data Explorer windows:

1. The *title bar* contains the following items (from left to right):
   - The menu button, which when pulled down, offers various window control options
   - The window name
   - The minimize button, which reduces the window and all of its secondary windows to a single icon
   - The maximize button, which enlarges the window to full screen size
2. The *menu bar* displays the titles of pull-down menus from which can choose options. When you choose a title, a pull-down menu appears.
3. A *pull-down* provides you with options you can select.

4. A *dialog box* or *information box* appears when Data Explorer requires more information in order to complete a task or when a user requests more information. A dialog box option is indicated by an ellipsis (...) at the end of its title.

Using the Mouse

In Data Explorer, the mouse is one of the primary input devices. Some operations can be done using the keyboard, but many rely on the mouse. The mouse-related terms used in this guide are:

Click	One press and release of a mouse button
Double-click	Two rapid presses and releases (clicks) of a mouse button
Triple-click	Three rapid presses and releases (clicks) of a mouse button
Drag	A press and hold on a mouse button. For example, with the button depressed, a selected item can be moved ("dragged") to another part of the screen by moving the mouse pointer to the desired location and releasing the mouse button.
Shift-Click	A press and release of the mouse button while depressing the Shift key.
Shift-Drag	A drag of the mouse pointer while depressing the Shift key.

Most of the Data Explorer operations use the left mouse button. The exceptions are the Online Help function and the image direct interactors, which use two or three mouse buttons.

Moving and Resizing Windows

You can move windows and adjust their sizes on the screen.

To move a window, drag the window name portion of the title bar to the desired screen location.

To resize a window, do one of the following:

- Drag on one of the horizontal borders. The window will shrink or expand vertically as you drag the border.
- Drag on one of the vertical borders. The window will shrink or expand horizontally as you drag the border.
- Drag on one of the corner borders. The window will shrink or expand both vertically and horizontally (but not necessarily in a uniform fashion) as you drag the border.

Selecting Pull-Down Menus and Pull-Down Menu Options

To use the mouse to see the list of options a pull-down menu offers, select the pull-down menu's title from the menu bar by moving the cursor to the appropriate title and clicking on it. The menu is displayed. To then select an option from the window, move the cursor to the desired option and click on the option.

Alternatively, select an option with the mouse by moving the cursor to the desired menu title and pressing and holding the mouse button. The pull-down menu appears. Then move the cursor to the desired option and release. Each option is highlighted as you move the cursor across it.

To select a pull-down menu with your keyboard, press the **Alt** key in conjunction with the underlined letter of the pull-down menu name (usually the first letter). For example, if you want to open the File menu (whose title appears in the window as **File**), press **Alt+F**.

Once the pull-down menu is displayed, you can select an option by pressing the key corresponding to the underlined letter of the desired option, or by directing the highlighted bar with the keyboard's up and down arrow keys. When the desired selection is highlighted, press the Enter key.

Some of the pull-down menu options can be selected without accessing the pull-down menus, using *accelerator keys*. Accelerator keys use the **Ctrl** key in combination with single keys to provide fast access to frequently used options. These keyboard options are displayed on the right side of the pull-down menu, across from the options that they access. Accelerator keys are effective only in the active window; that is, the mouse cursor must be in the window in which the desired pull-down option is located. To use an accelerator key to select a pull-down option, press the **Ctrl** key in conjunction with the appropriate letter. For example, the **Save** option in the **File** pull-down menu can be invoked by pressing the **Ctrl** key and the **S** key at the same time. A summary of the available accelerator keys and their functions is provided in Appendix G, "Accelerator Keys" on page 315.

For information on each window's menu bar and pull-down options, refer to:

- "VPE Window Menu Bar" on page 156
- "Control Panel Menu Bar" on page 162
- "Image Window Menu Bar" on page 165
- "Colormap Menu Bar" on page 168

Selecting and Deselecting Items with the Mouse

In the Visual Program Editor (VPE) and Control Panel windows, you use the mouse to select various items, such as items in a list, tool names, tool icons, and interactors. To select any of these items, simply click on the item.

In general, you can deselect an item by clicking on it again. The exceptions are the tool icons and interactors, for which you must either click on another part of the canvas, or shift-click on the item.

Selecting a Choice in an Option Box

An option box contains a list of choices, but usually displays only the one currently selected. Option boxes are used throughout the Data Explorer interface. For example, the Selector interactor, illustrated in Figure 31 on page 64, can be displayed as an option box.

Selecting a choice from this box is similar to selecting options from pull-down menus. To display the possible choices, click on the tab on the right side of the option box. With the list of choices displayed, click on your desired selection. The list of options disappears, and the option box is updated with the new selection.

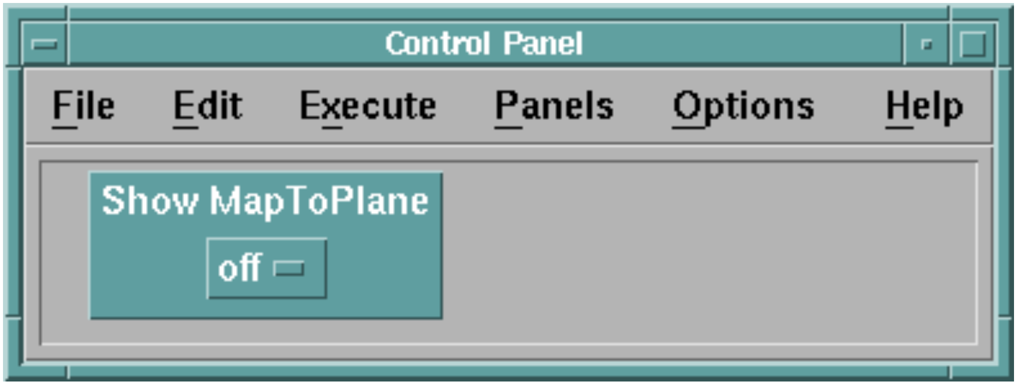


Figure 31. Example of an Option Box

Alternatively, you can select an option with the mouse by moving the cursor to the desired option box and pressing and holding the mouse button. The list of options appears. Then move the cursor to the desired option and release. Each option is highlighted as you move the cursor across it.

Some of the options are accessible using accelerator keys. This is indicated to the right of the option in the options list, much like it is in the pull-down menus.

Editing Text Fields

Many of the dialog boxes in Data Explorer include text fields that you can change. You can place a text cursor in the box by clicking on the box. With the cursor in the box, you can use the keyboard to alter the text. The number of times you click on the text box depends on how you want to edit it:

Single Click	Places the text cursor at the point on which you clicked in the existing text field. All text you type is then inserted in the field, after the selected point. You can also use the Delete and Backspace keys.
Double Click	Places the text cursor at the start of the word on which you clicked in the existing text field, and highlights that word. When you type, the highlighted text is replaced with the new text you enter. Pressing the Backspace key deletes the highlighted text.
Triple Click	Highlights the entire text field. When you type, the entire field is replaced with the new text you enter.
Drag	Highlights the portion of the text field that you drag the cursor over. When the mouse button is released, the text remains highlighted. Typing anything replaces the highlighted text with what you type; pressing the Backspace key deletes the highlighted text.

Data Explorer allows you to copy and paste text within text fields. To copy text, select the desired text by double-clicking, triple-clicking, or dragging the cursor over it with the left mouse button. This action selects and copies the text. To paste the selected text in a text field, position the mouse cursor at the point you want to insert the text, and click on the center mouse button. The pasted text is inserted at the mouse cursor position.

If the amount of text entered into a text field cannot be fully displayed, you can use the keyboard to scroll the cursor through the text. Pressing the left or right arrow key moves the cursor one character to the left or right, respectively. Also, pressing the Control key (**Ctrl**) at the same time as either the left or right arrow key moves the cursor one *word* to the left or right, respectively. The Home key will move the cursor to the beginning of the text field. The End key will move the cursor to the end of the text field.

Working with Windows

Your window manager allows you to:

- Stack and raise windows
- Minimize windows
- Maximize windows
- Close windows

These functions can be invoked using the menu, minimize, and maximize buttons on the title bar of Data Explorer windows. You can use the X Window System or the appropriate window system manager to customize the way these options work. For more information, see *IBM AIXwindows User's Guide* or another appropriate window system overview.

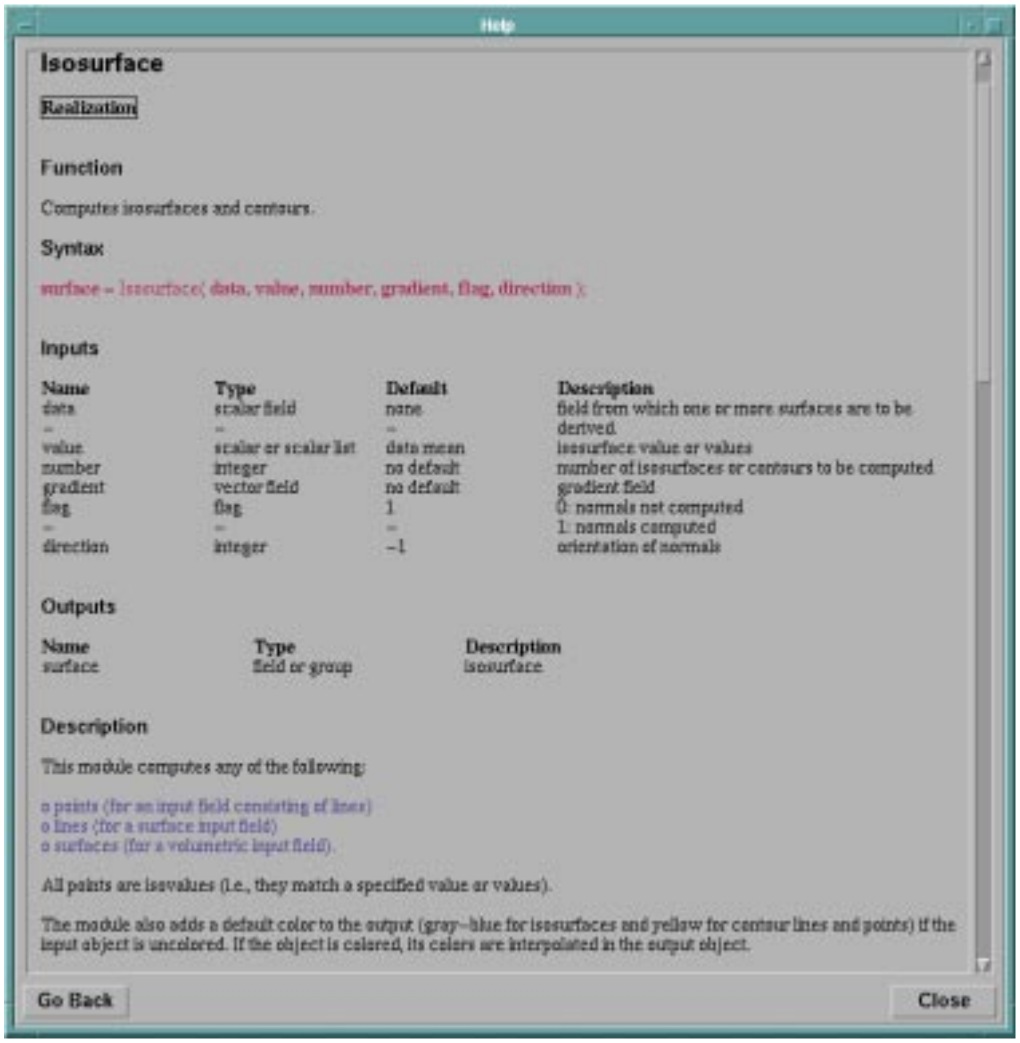
5.3 Using Online Help

The online **Help** facility offers the following options in its pull-down menu:

- **Context-Sensitive Help** can help you with a specific tool or feature of Data Explorer's graphical user interface. After you select this option and the cursor has changed to a question mark, click on the item you want help with. For example, try clicking on a particular module in the VPE window or on an interactor in a Control Panel.

  Note: You can also position the mouse cursor over an object and press the **F1** key to select an object in any window, including the options in a pull-down menu and icons on a VPE canvas..
- **Overview (of Window)...** gives an introductory description of the window from which this option was requested.
- **Table of Contents...** presents a list of the main topics and subtopics of the Data Explorer user manuals. Select the desired item by clicking on its box once.
- **Using Help...** explains how to use the online facility itself.
- **Product Information...** gives the version of Data Explorer that is currently running.
- **Technical Support** gives information on how to get technical support.
- **Tutorial...** presents a list of tutorial topics, which can be accessed directly.
- **Application Comment** presents a comment on the currently loaded visual program (if one was supplied by the user who created the program).

Figure 32 on page 66 depicts a sample Help window.



Isosurface

Realization

Function

Computes isosurfaces and contours.

Syntax

```
surface = Isosurface( data, value, number, gradient, flag, direction );
```

Inputs

Name	Type	Default	Description
data	scalar field	none	field from which one or more surfaces are to be
–	–	–	derived
value	scalar or scalar list	data mean	isosurface value or values
number	integer	no default	number of isosurfaces or contours to be computed
gradient	vector field	no default	gradient field
flag	flag	1	0: normals not computed
–	–	–	1: normals computed
direction	integer	–1	orientation of normals

Outputs

Name	Type	Description
surface	field or group	isosurface

Description

This module computes any of the following:

o points (for an input field consisting of lines)
o lines (for a surface input field)
o surfaces (for a volumetric input field).

All points are isovalues (i.e., they match a specified value or values).

The module also adds a default color to the output (gray–blue for isosurfaces and yellow for contour lines and points) if the input object is uncolored. If the object is colored, its colors are interpolated in the output object.

Figure 32. Sample Help Window

In addition, an HTML version of the documentation is available. Point your browser at `/usr/lpp/dx/html/index.htm`.

For Future Reference

- In the **Help** window you can directly access any topic, subtopic, *sub*subtopic, or related item that is "boxed", simply by clicking on the box.
- If the amount of information exceeds the length of the displayed **Help** window, use the vertical scroll bar on the right side of the window.
- To return to a topic you have viewed during the current session of visual program, move the mouse cursor into the **Help** window and press the *right-hand* button: A list of viewed Help topics appears. Keeping the mouse button depressed, move the cursor to the desired topic in the list and then release the button.
- To return to the previous Help topic, click on **Go Back** at the bottom of the **Help** window.
- To exit online Help, click on **Close** at the bottom of the **Help** window.

User-Defined Help Files

Visual programmers can create online documentation for their visual programs and Control Panels. A user can access the comments for the visual program from any primary window, and can access the comments for a Control Panel from the Control Panel with which the comments are associated. For information on how to add this documentation to your own visual program, see "Adding Comments to a Visual Program" on page 114 and "Customizing a Control Panel" on page 133.

To access comments for the visual program, select the **Application Comment** option from the **Help** pull-down menu. A dialog box opens with the comments.

Note: If there are no comments associated with the visual program, this menu option is grayed-out. Also, you cannot modify the comments when using this option—only view them.

The user can also access the comments for the visual program from the **Open...** or **Load Macro...** file selection dialog boxes, even before the visual program is opened. For information on how to do this, see "Restoring a Previously Created Program" on page 118.

To access comments for a specific Control Panel, select the **On Control Panel** option from the **Help** pull-down menu in the Control Panel about which you want to learn. A dialog box opens with the comments.

Note: If there are no comments associated with the Control Panel, this menu option is grayed-out. Also, you cannot modify the comments when using this option—only view them.

5.4 Executing a Visual Program

After you set up a visual program in the VPE window and build any desired Control Panels (or after you open an existing visual program file), you can execute the visual program program. The resulting image appears in the Image window. This section explains how to execute a visual program. Section 6.1, "Using the Image Window" on page 74 describes how to manipulate images in the Image window by using direct interactors. You can also manipulate images in the Image window using interactors in the Control Panels, the Colormap Editor, and the Sequencer. (For more information on these tools, see 7.1, "Using Control Panels and Interactors" on page 128, "Colormap" on page 84, and "Sequencer" on page 297 in *IBM Visualization Data Explorer User's Reference*.)

You can execute the visual program from the **Execute** menu of a VPE window, a Control Panel, an Image window, or a Message window. The options are the same in all four windows. When a visual program is executed, an Image window is created if one is not already open. (You can also control execution using the Execute module. See *IBM Visualization Data Explorer User's Reference*.)

Note: If the **Execute** options are grayed-out, your workstation may not be connected to the server. For information about connecting to the server, see 9.3, "Connecting to the Server" on page 183.

Although you can initiate execution from the Control Panel, VPE, Image, and Message windows, you may find it more efficient to execute your visual program through the Control Panel menu (if you are using a Control Panel). This efficiency is due to the ease with which you can change inputs with interactors and initiate execution.

You can choose one of four options from the **Execute** menu when executing a visual program. Select the first option, **Execute Once**, to execute the program once, using the values currently set in the interactors. If you change any interactor values after execution, the visual program does not automatically execute; you must again choose an option from the **Execute** menu to execute the altered program.

Choosing the **Execute on Change** option causes the visual program to execute every time you change an interactor setting. If you change values faster than Data Explorer can generate images, the system executes the program as quickly as possible, always using the current settings at the time an execution cycle begins. If you modify your visual program while **Execute on Change** is enabled, then the option automatically becomes disabled. After the changes are made, you can reenable it.

Choosing **End Execution** while the visual program is executing causes execution to stop after the currently executing module.

The final **Execute** menu option is **Sequencer**. If you select this option, and a Sequencer tool is present in the visual program, the Sequencer appears (see “Using the Sequencer” for more information). While the Sequencer runs, you can change interactor settings, and those changes are reflected in subsequent frames generated by the Sequencer. The **Execute Once** and **Execute on Change** options are grayed out when the Sequencer is running, but when you pause the Sequencer, you can use those two options to explore the particular frame the Sequencer paused on.

While the visual program is executing, the **Execute** option on the menu bar is highlighted. It remains highlighted until execution is finished. If **Execute on Change** is selected, the **Execute** option on the menu bar is highlighted with one color during execution, and another color outside of execution cycles.

Using the Sequencer

The Sequencer allows you to “animate” a visual image and is very easy to use. The process is rather like running a video cassette tape: You can play it forward or backward, stop it, pause, and so on. The Sequencer Control panel consists of 8 buttons as shown in Figure 33.

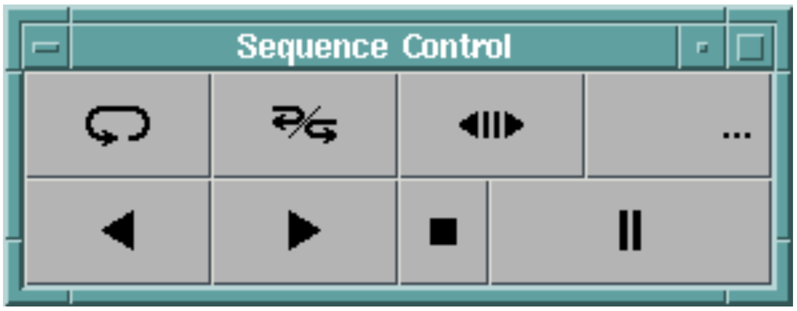


Figure 33. Sequence Control Panel. The first two buttons at top left are Loop and Palindrome. The others are: Step (◂||▸), Counter (...), Back (◂), Forward (▸), Stop (▪), and Pause (||).

The ► button starts the animation sequence and plays forward. The ◄ button plays the sequence in the opposite direction.

The ■ button stops the animation and resets the animation to the beginning of the sequence, while the || button pauses the animation at the current frame.

The **Loop** button causes the animation to loop; that is to go from beginning to end, reset to beginning, play to end, and so on until terminated by either pause or stop.

The **Palindrome** button causes the sequence to be played from beginning to end, and then from end to beginning.

Loop and **Palindrome** can be pressed simultaneously, resulting in an continuous forward and reverse animation.

The ◄||► button causes the behavior of the ► and ◄ buttons to become single-step mode. Each time one of these buttons is pressed, the animation advances one frame in the specified direction.

The **...** button opens the Frame Control dialog. The Frame Control dialog box (see Figure 34) is used to specify the first, "next," and last (end) frames, the number of frames, and the increment between successive frames.

If a frame is being displayed, the current frame number appears in the Frame Control dialog box, next to the word "Current," and a corresponding colored marker is shown on the slide bar. A colored marker indicating the position of the next frame is also shown. Black markers indicate the positions of *start* and *end* relative to the next range of min to max.

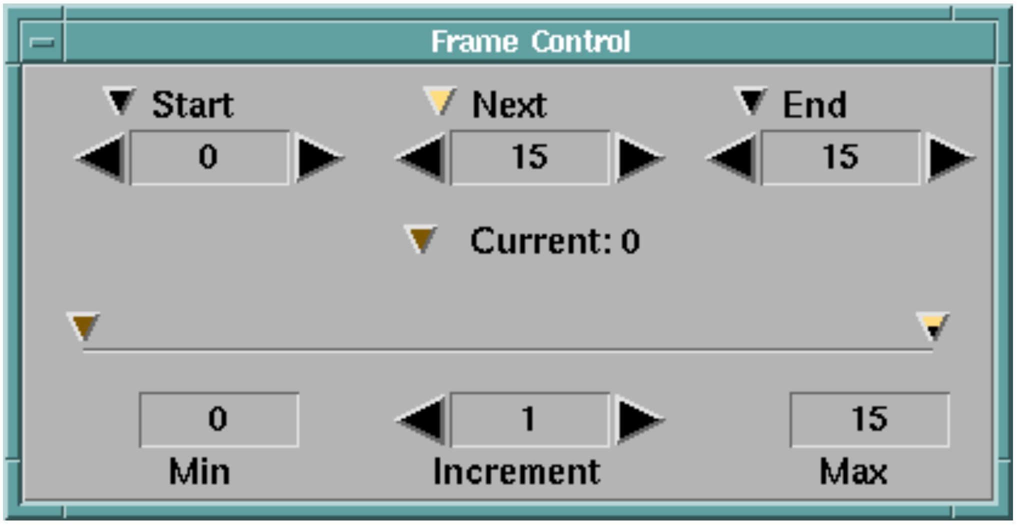


Figure 34. Sequencer Frame Control Dialog Box

Values for the Start, Next, and End frames are set by:

- Entering a value in the text field of the stepper buttons
- Using the stepper controls
- Moving the position marker.

Start The starting value for the sequence. By default, set to the value in the Min field, in a new program. To change the Start field, use the stepper controls or select the field and enter the new value or use

	the Start marker. If you change the value in the Min field, then the Start field is set to that new value. If you are working with a saved program, then the Min and Start fields are set to the values that were saved.
End	The ending value for the sequence. By default, set to the value in the Max field, in a new program. To change the End field, use the stepper controls or select the field and enter the new value or use the End marker. If you change the value in the Max field, then the End field is set to that new value. If you are working with a saved program, then the Max and End fields are set to the values that were saved.
Current	Displays the current frame number.
Increment	By default, set to 1. To change the increment, use the stepper controls or select the field and enter the new value.
Next	By default, set to the value in the Start field, in a new program. You can set the Next field to any value between the Start and End values: the Sequencer will begin running at that value. (When the Sequencer is in loop mode, subsequent loops begin at the value in the Start field.)
Min and **Max**	Specify the allowed range of sequence values. These are text fields that can be altered. Data Explorer ensures that the value in the Start field is greater than or equal to the Min value, and the value in the End field is less than or equal to the Max value.

Note: If the images change more quickly than you would like, use the **Throttle...** option (see "Changing the Rate of Frame Display: Throttle..." on page 93).

Using a Data-Driven Sequencer

The Sequencer can be *data driven*, meaning that its minimum, maximum, and step values can be set by connecting the output of a tool to the input of a Sequencer in the VPE or by a value typed into the Sequencer's Configuration dialog box, rather than by using the **Frame Control** panel.

If the Sequencer is data driven, then the information transmitted by connection or set in the Configuration dialog box overrides values set in the **Frame Control** panel.

A data-driven Sequencer allows you to create visual programs that will work with a variety of input data sets without your having to reset Sequencer attributes. For example, if the Sequencer minimum is set to zero and its maximum to the number of steps in a series, it can be used to drive the Select module to select each member of the series in turn.

The inputs are summarized in the corresponding module description in *IBM Visualization Data Explorer User's Reference*.

Each time an input to a data-driven Sequencer is changed (for example, by importing a new data set) the Sequencer is reexecuted, updating its attributes.

Error Messages

If Data Explorer encounters an error in your visual program while executing it, an error message is displayed in the Message window (see 8.2, “Using the Message Window” on page 174). The name of the tool in which the error occurred is shown in the window. Pull-down menu options enable you to quickly locate the tool that caused the error.

The title of the tool icon in the visual program that caused the error is displayed in a different color in the VPE until you execute the program again. When the error occurs, execution stops only in the path where the error is; other paths continue.

Chapter 6. Graphical User Interface: Important Windows

This chapter discusses how to use:

- The Image window
- The Visual Program Editor
- The Colormap Editor.

6.1 Using the Image Window

Once an image appears in the Image window, it can be altered and manipulated in many ways. This section describes how to manipulate the image directly, using the interactor features accessed through the **Options** menu in the Image window.

Notes:

1. These features are enabled *only* if you used the Image tool in your visual program to render the image. If you used another tool, such as **Display**, you will not be able to use many of the Image window options. The Image tool is found in the **Rendering** category. See "Image" on page 160 in *IBM Visualization Data Explorer User's Reference* for more information.
2. It is also possible to control many characteristics of the Image widow by using input parameters to the Image tool. These parameters are hidden by default but can be accessed with the **Expose** button in the Image tool's Configuration dialog box (which itself is accessed by selecting the **Configuration** option in **Edit** pull-down menu of the Visual Program Editor) For more information, see Image in the *IBM Visualization Data Explorer User's Reference*.

Controlling the Image: View Control...

You can change your view of an object in the Image window by using the **View Control...** option of the Image window **Options** menu. The way an object appears in the Image window is controlled by a *camera*. The camera specifies the *look-from point*, the *look-to point*, the field of view, the size and shape of the window, and whether the rendering method is orthographic or perspective.

If you are using orthographic projection, the field of view is controlled by a parameter called *width*, which specifies the width of the Image window in world coordinates. For perspective projection, the field of view is controlled by the camera look-from position and the view angle of the camera. The differences between these two projection methods is discussed in "Changing the Projection Method" on page 76.

You can change the camera settings with *direct interactors*, which let you use the mouse in conjunction with dialog box options to manipulate the view of the object directly, and which provide immediate visual feedback for your actions. Or, you can specify precise values for the camera settings, as described in "Precise Camera Settings" on page 82.

You control your view of the object by selecting one of the control *modes*. Each mode allows you to control different aspects of your view. When you select this option, the **View Control...** dialog box appears. The basic configuration of this dialog box is illustrated in Figure 35 on page 75.

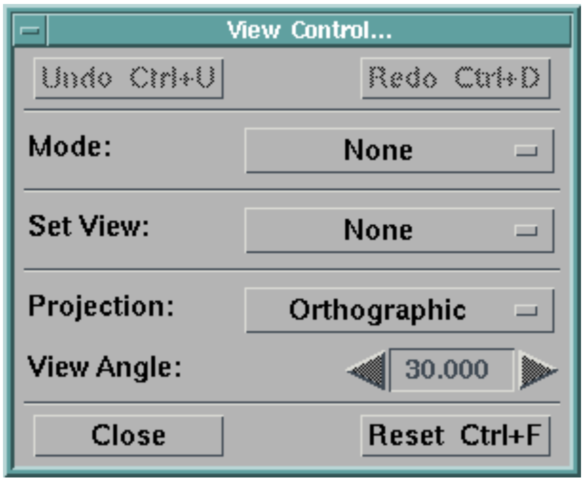


Figure 35. View Control Dialog Box

Depending on the view mode selected, the appearance of the dialog box changes, automatically adding controls particular to the current mode.

Accelerator keys are associated with many of the modes, so that it is not even necessary to open the **View Control...** dialog box to access these view control modes. Figure 36 lists the modes with their respective accelerator keys.

None	
Camera	Ctrl+K
Cursors	Ctrl+X
Pick	Ctrl+I
Navigate	Ctrl+N
Pan/Zoom	Ctrl+G
Roam	Ctrl+W
Rotate	Ctrl+R
Zoom	Ctrl+Z

Figure 36. View Control Modes with Accelerator Keys

You can change your view of the image by:

- Changing your viewing direction
- Changing the projection method
- Rotating the object (**Rotate** mode)
- Zooming in or out of the image (**Zoom** mode)
- Changing the look-to point (the center) of the image (**Roam** mode)
- Panning and zooming in or out of the image (**Pan/Zoom** mode)
- Moving the camera within the scene (**Navigate** mode)
- Specifying precise values for the camera settings (**Camera** mode)
- Setting probe points in the image (**Cursors** mode)
- Picking objects in the image (**Pick** mode)
- Resizing the image
- Resetting previous views (**Undo** and **Redo** buttons).

To select a mode, use its accelerator key, or do the following:

1. Click on the **Mode** option box. A menu of the mode options is displayed.

2. Click on the desired option. The option box displays the name of the new mode. The dialog box also displays any additional controls for the new mode.

The different modes are discussed in more detail in subsequent sections. Most of the modes use two or three mouse buttons to manipulate the view. A diagram is provided in the margin to the left of each section to illustrate the mouse button functions, the mode, and the accelerator key.

Some of the modes use an overlay, which provides visual feedback before any action is taken to change the image. This temporary overlay (typically an axes diagram, rectangle, or brick) is laid over the image. As you drag with the mouse, the overlay changes accordingly. The image does not change until you release the mouse button, at which time it is updated according to the action you selected.

Note: If **Execute on Change** is enabled, Data Explorer does update the image *while* you perform rotation actions, and while you place probes in **Cursors** mode.

Changing Your Viewing Direction

Data Explorer allows you to choose from several defined viewing directions. To change the view:

1. Select the **Set View** option box in the **View Control...** dialog box. While the mouse button is pressed down, an option box listing different view options appears. These choices are illustrated in Figure 37.

```
None
Top
Bottom
Front
Back
Left
Right
Diagonal
Off Top
Off Bottom
Off Front
Off Back
Off Left
Off Right
Off Diagonal
```

Figure 37. Set View Option Box

Choose to view the object from the top, bottom, front, back, left, right, or diagonal view. Because a head-on view of an object tends to detract from the 3-D quality, **Off** options are provided to skew the image slightly. To select an option, move the mouse so that the desired option is highlighted, then release the mouse button. The image is redisplayed with the new view.

Changing the Projection Method

Data Explorer provides two methods of image projection. These methods make it possible to map a three-dimensional image onto a 2-dimensional screen. The two methods are:

Perspective Perspective projection simulates normal camera and human visual systems, thereby providing realistic rendering of objects. While it is realistic, it does not preserve the exact shape and measurements of the object, and parallel lines usually do not project as being parallel.

Orthographic Orthographic projection provides a less realistic view of an object than perspective projection. However, orthographic projection does preserve exact scale measurements and parallel lines. One way to think about orthographic projection is that it is as if the distance between the front and back of an object is small, relative to the distance between the object and the camera. Orthographic is the default projection method. It is quicker to render an object in orthographic projection than in perspective.

The differences between these two methods become evident when using the view controls in certain modes. For example, the **Zoom** mode behavior depends on the projection method, as does the behavior of the 3-D cursor (**Roam** and **Cursors** modes). The differences are discussed in the sections that discuss these modes. For more information about these projection methods, consult a computer graphics text.

You can select the projection method by using the **Projection** option box on the **View Control...** dialog box.

Setting the View Angle: If you select **Perspective**, then you can also specify the view angle (in degrees). The vertex of the view angle is located at your camera, and the end points are the left and right sides of the image area. Thus, the wider the viewing angle, the more image space you can fit in your viewing area.

Specify the view angle by adjusting the **View Angle** stepper in the **View Control...** dialog box. Unless you are in **Navigate** mode, the camera position is adjusted so that the size of the object in the image area remains unchanged. While in **Navigate** mode, changing the view angle does not change the camera position.

While you are using orthographic projection, the **View Angle** stepper is disabled. In orthographic projection, you can specify the camera width. This is discussed in “Precise Camera Settings” on page 82.

Note: Changing the projection method or view angle does not automatically initiate reexecution of the visual program. To see the effects of your changes, you must select **Execute Once** or **Execute on Change**.

Rotating the Object

To rotate the object you are viewing:

1. Select **Rotate** from the **Mode** pull-down box or use the **Ctrl+R** accelerator key.
2. A set of 3-D axes appears in the lower right corner of the Image window. Also, if you have enabled the **Display Rotation Globe** option (in the **Options** pull-down menu), or you do so now, a wire-frame globe appears in the lower left corner. If you press the *right* mouse button, you can drag the mouse to rotate the object clockwise (CW) and counter-clockwise (CCW), with the image center as the point of rotation. If you press the *left* or *center* mouse button, you can use the mouse as a track ball and rotate the object in all three dimensions. The mode of rotation is determined by the mouse button you use.

As you rotate the object, the set of axes and the wire-framed globe (if enabled) also rotate, providing instant feedback to your actions.

Note: If you do not have **Execute on Change** enabled, only the set of axes and the globe are updated as you move the mouse, providing preliminary feedback to your rotation actions. If **Execute on Change** is enabled, then the object rotates as you move the mouse.

3. When you release the mouse button, the object is rotated according to the new position of the axes and globe.

   Note: You can also enable **Rotate** mode by using the **intrctnMode** parameter to the Image tool (see Image in *IBM Visualization Data Explorer User's Reference*).

The point around which the object rotates is the look-to point, which you can change by selecting **Roam** (see "Changing the Look-to Point" on page 79). Alternatively, you can select **Pan/Zoom** in the **Mode** pull-down menu (see "Zooming into and out of the Image").

Zooming into and out of the Image

To zoom in or out (relative to the center of the Image window):

1. Select the **Zoom** mode in the **Mode** option box, or use the **Ctrl+Z** accelerator key.
2. To zoom in, press the *left* mouse button; to zoom out, press the *right* mouse button. Drag the mouse in the image area. This causes an overlay image of a rectangle to appear. As you move the mouse pointer away from the center of the window, the rectangle enlarges. As you move the mouse pointer towards the center of the window, the rectangle shrinks. Releasing the mouse button causes the image to zoom in or out, depending on which mouse button is pressed. If the left button (zoom in) is pressed, the portion of the image inside the rectangle is enlarged to fill the Image window. If the right button is pressed (zoom out), the image displayed in the Image window is reduced to the size of the rectangle.

   Note: You can also enable **Zoom** mode by using the **intrctnMode** parameter to the Image tool (see Image in *IBM Visualization Data Explorer User's Reference*).

For both orthographic and perspective projection, zooming in makes the object appear larger in the Image window, while zooming out makes it appear smaller. However, the way that this is accomplished is different depending on which projection method is selected:

- While using perspective projection, zooming in or out changes the view angle, without changing the look-from point.
- While using orthographic projection, zooming in or out adjusts width of the image. The camera look-from point does not change, because with orthographic projection the look-from parameter merely specifies a direction, not a location in space.

For more information about the projection methods, see "Changing the Projection Method" on page 76.

Changing the Look-to Point

Data Explorer maintains the look-to point at the center of the Image window. You can use the **Roam** mode to change the look-to point.

To change the look-to point:

1. Select the **Roam** mode by using the **Mode** option box or the **Ctrl+W** accelerator key.
2. A wire-frame box appears around the image. The current look-to point is marked by a small square box. To move the look-to point, select the point by pressing the *left* mouse button and holding the mouse pointer on it. You can then move the cursor by dragging the selected point inside the box. When the left mouse button is depressed on the point, the mouse cursor disappears (the look-to point remains) and the three projections (one for each axis) appear inside the wire-framed box as dots, similar to the 3-D cursor function illustrated in Figure 40 on page 86. When you release the mouse button, the location of the point becomes the new look-to point, and the image view is updated.

   Alternatively, double-clicking on a point in the Image window changes the look-to point to the position on which you double-click. You can set the look-to point to a location outside the current boundary box of the object using this method.

   Note: You can also enable **Roam** mode by using the **intrctnMode** parameter to the Image tool (see Image in *IBM Visualization Data Explorer User's Reference*).

The directions of the axes are indicated by a wire-frame 3-D axes diagram in the lower right-hand corner of the Image window. To move the look-to point, the movement of the mouse must be along the same direction as one of the axes. If you move the mouse in a direction that does not correspond to any of the axes directions, the point does not move. Because of this, the movement of the look-to point is constrained to six directions (i.e., the positive and negative directions for each of the three axes).

Note that since perspective projection does not preserve parallel lines, the directions in the axes diagram do not necessarily correspond to the direction that the point moves in the Image window. However, these axes do correspond with the values of the coordinates.

While using orthographic projection, the movement of the point in the Image window corresponds with the directions of the axes diagram.

The **Constraints** option appears in the **View Control...** dialog box when you select **Roam** mode.

You can further restrict the movement of the look-to point by using the **Constraints** option. This allows you to constrain movement of the point to two directions (i.e., the positive and negative directions of a particular axis). Select an axis to restrict movement to by clicking on the **Constraints** option box and selecting the desired axis (X, Y, or Z). Once this is done, you are able to move the look-to point along only the selected axis, while the values of the other two axes remain constant.

You may want to position the cursor in two steps, using different views for each step. For example, you can position the *x* and *y* coordinates using the front view, and the *z* coordinate using the side view.

To release the constraints on an axis, choose the **None** option.

While in **Roam** mode, the center and right mouse buttons have the same functions as they do in **Rotate** mode.

Panning and Zooming into and out of the Image

You can zoom into or out of the image and change the look-to point at the same time, using the **Pan/Zoom** mode. In this mode, the left and right mouse buttons work the same as they do in **Zoom** mode, except that the point on the image where you initially click becomes the new look-to point, and therefore the center of your zoom action. As you move the mouse pointer away from the point where you initially clicked, the rectangle enlarges. As you move the mouse pointer towards the point where you initially clicked, the rectangle shrinks. For information about the **Zoom** mode, see "Zooming into and out of the Image" on page 78.

To select **Pan/Zoom** mode, select its option from the **Mode** option box, or use the **Ctrl+G** accelerator key.

Note: You can also enable **Pan/Zoom** mode by using the **intrctnMode** parameter to the Image tool (see Image in *IBM Visualization Data Explorer User's Reference*).

Navigating in a Scene

You can move your camera within the scene in all directions using the **Navigate** mode. As well as being able to change your view direction like the other modes, **Navigate** also lets you move *through* an object and view it from the inside out.

When you use *Navigate* mode, the camera is automatically changed to perspective rendering.

You can also move your camera in one direction while looking in another direction. For instance, you can move forward past the left side of an object while looking to the right, enabling you to see the object as you move past it.

Unlike the **Roam** and **Rotate** modes, which appear to adjust the object to change your view, or the **Zoom** mode, which appears to adjust the camera lens to change your view, the **Navigate** mode appears to actually move your camera around the scene, while the objects you are viewing remain stationary.

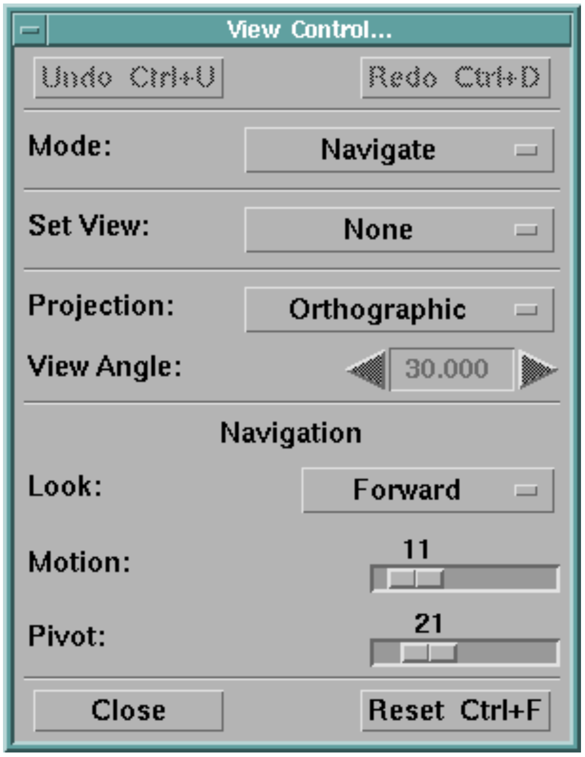
Note: Because Data Explorer does not support perspective rendering of volumes, you cannot navigate software-rendered volumes.

To move your camera within the scene:

1. Select the **Navigate** mode from the **Mode** option box. This adds some controls, illustrated in Figure 38 on page 81, to the **View Control...** dialog box.
2. The mouse buttons control the navigate motion of the camera, while the setting of the **Look** option box controls the direction the camera is pointing. The **Look** options are illustrated in Table 4 on page 81.

   Note: You can also enable **Navigate** mode by using the **intrctnMode** parameter to the Image tool (see Image in *IBM Visualization Data Explorer User's Reference*).

   With **Look** set to **Forward**, you have more control over the trajectory the camera moves along in the scene. To move forward, click on the left mouse button. To move backward, click on the right mouse button. The position of the mouse pointer on the image indicates the angle at which you travel; that is, the point where the mouse pointer is located moves more towards the center of the image area.



Mode: Navigate
Set View: None
Projection: Orthographic
View Angle: 30.000
Navigation
Look: Forward
Motion: 11
Pivot: 21

Figure 38. Navigate Portion of the View Control Dialog Box

Table 4. Look Option Menu
Forward
45° Left
45° Right
45° Up
45° Down
90° Left
90° Right
90° Up
90° Down
Backward
Align

If the camera is pointing forward, you can use the center mouse button to change your direction. Clicking on the center mouse button adjusts your direction (and view of the object) toward the position of the mouse pointer in the image area.

With the exception of **Align**, all of the options in the **Look** option box affect only the direction the camera is pointing. **Align** adjusts the current navigate direction (the forward direction) to be the same as the current **Look** value. For example, if the camera is pointing 45° to the right of the current navigate direction, **Align** changes the navigate direction to be 45° to the right of its current value, thereby aligning travel direction with camera direction.

Note: Switching to another mode (such as **Rotate**) from **Navigate** while the Look value is something other than Forward, automatically aligns the navigate direction with the camera direction.

If the camera is pointing in any direction but forward, you can still use the left and right mouse buttons to move forward and backward; the trajectory of your movement is the same as it was when you last traveled forward. In this case, the center mouse button has no effect.

3. The **Navigate** mode uses two values to control its precision. These are controlled by the two sliders on the **View Control...** dialog box. They are:

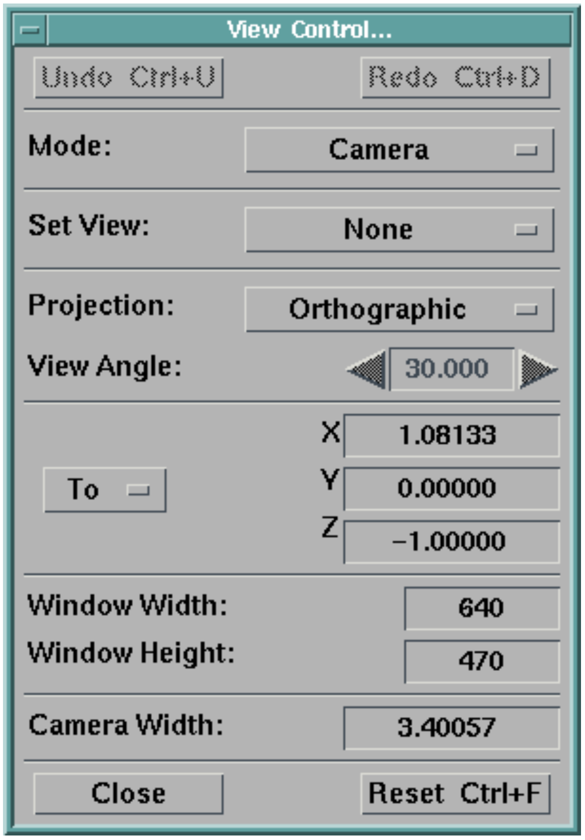
Motion Controls the speed of execution as you move the camera forward and backward. The higher the number, the faster the speed, and the faster your camera appears to move. In addition to using the slider, you can use the up- and down-arrow keys to adjust this value (be sure the cursor is in the Image window).

Pivot Controls the increment of the turn angle when you turn using the center mouse button. The higher the number, the more drastic the turn. In addition to using the slider, you can use the left- and right-arrow keys to adjust this value (be sure the cursor is in the Image window).

Precise Camera Settings

In addition to using direct interactors to change camera settings, Data Explorer lets you specify exact values to the camera for more precise results. You can also use this feature to learn the exact camera settings that result from your use of direct interactors. These settings correspond to parameters of the Camera module. (See "Camera" on page 49 in *IBM Visualization Data Explorer User's Reference*.)

To access the camera settings, select the **Camera** mode using the **Mode** option box in the **View Control...** dialog box or the **Ctrl+K** accelerator key. The camera controls appear on the **View Control...** dialog box, as illustrated in Figure 39 on page 83.


View Control...

Undo Ctrl+U   Redo Ctrl+D

Mode: Camera

Set View: None

Projection: Orthographic

View Angle: 30.000

To   X 1.08133   Y 0.00000   Z −1.00000

Window Width: 640

Window Height: 470

Camera Width: 3.40057

Close   Reset Ctrl+F

Figure 39. Camera Settings Portion of the View Control Dialog Box

Note that these camera controls are not direct interactors, so changing their values does not cause the visual program to reexecute automatically. To see the effect of your changes, you must specify **Execute** or **Execute on Change** in the VPE, Control Panel, or Image window.

Changing the Look-to Point, Look-from Point, and the Up Vector: You can adjust your view of the image by specifying values for these settings:

look-to point vector	The point around which the displayed image is centered. The point is specified as a vector in world coordinates.
look-from point vector	The position of the camera. The point is specified as a vector in world coordinates.
up vector	The rotation (tilt) of the camera. Only the direction of this vector is important, not the magnitude. For instance, if you are looking at the object from a positive *z* direction, an up vector with a negative value for *x* tilts the camera counterclockwise.

The option box near the center of the **View Control...** dialog box (displaying **To:** by default and in Figure 39) allows you to select the vector values to be displayed.

To change the value of one of the vectors, do the following:

1. Select the vector to change using the option box. The choices in the option menu are **To**, **From**, and **Up**. The current values for the vector you choose are displayed in the X, Y, and Z fields to the right of the options.

2. Change each field to the new value. Do this by clicking on the field, typing in the new number, and pressing the enter key.
3. Repeat these steps for each vector you want to change.

Remember, to see the results of your changes, you must specify **Execute** or **Execute on Change**.

Changing the Size of the Image Window: Data Explorer lets you specify the exact height and width of the image display area in the Image window, in pixels. When you change the size of the image display area, the image is resized accordingly.

To change the width and height of the image display:

1. Click on the **Window Width** field of the **View Control...** dialog box illustrated in Figure 39 on page 83.
2. Type a new value, specified in pixels, for the width. Press the enter key.
3. Repeat steps 1 and 2 for the **Window Height** field.

To see effect of the changes, specify **Execute** or **Execute on Change**.

Setting the Camera Width: If you select **Orthographic** projection, then you can also specify the width of the field of view. The larger the width, the smaller the object appears.

Specify the width by clicking on the **Camera Width** field, typing a new value (in world coordinates), and pressing the Enter key.

While you are using perspective projection, the **Width** field is grayed out.

Resizing the Image

To resize the image, simply resize the image window by dragging its borders to shrink or expand the window. The image displayed inside the window is resized accordingly. Note that the size of the object in the Image window is controlled only by the width of the window.

You can specify an exact Image window size by using **Camera** mode, as described in "Precise Camera Settings" on page 82.

For more information on resizing windows, see "Moving and Resizing Windows" on page 62.

Restoring Images

Data Explorer remembers the 10 most recent camera configurations. You can undo the most recent actions in the **View Control...** dialog box by selecting the **Undo** option box or by using its accelerator key, **Ctrl+U**. You can continue to revert to previous configurations until you have reached the first configuration in memory; after that, the **Undo** option is disabled (grayed out).

As you revert to previous camera configurations (by **Undo**ing actions), you can still restore them by using the **Redo** button, providing that you have not performed any other actions since the last **Undo**. The accelerator key for **Redo** is **Ctrl+D**.

Note: The **Redo** button is disabled as soon as you change the camera configuration by a means other than **Undo**. Any configurations that have been stored and "undone" are discarded.

Resetting the Camera: To return the image to a “front” view that includes the entire object, select the **Reset** option of the **View Control...** dialog box, or use the **Ctrl+F** accelerator key.

You can also use this option if you bring in a new data set and want to create a new camera appropriate for the data.

Note: You can also reset the camera by using the **resetCamera** parameter to the Image tool (see Image in *IBM Visualization Data Explorer User’s Reference*).

Using Probes (Cursors)

A probe is a list of one or more vectors that represent points in the image. You can use them with Data Explorer modules that accept vectors as input, such as ClipPlane and Streamline.

After you execute a visual program and have an image in the Image window, you can modify the visual program to include a **Probe** or **ProbeList** tool from the **Special** category. The probe tool accepts input from the 3-D cursor tool, specifying the points to use as vectors for input into another tool. The **Probe** tool accepts one point as its input; the **ProbeList** tool accepts multiple points as input.

To use probes to select points for input into tools:

1. Execute a visual program to produce an image in the Image window.
2. In the VPE, place one or more probe tools from the **Special** category in the visual program, connecting them to the tools for which you want to provide input.

   The probe icons are numbered as you place them on the canvas. For example, the first probe icon you place is labeled “Probe_1,” the second “Probe_2,” and so on. You can change the label of the icon by using its Configuration dialog box.
3. In the **View Control...** dialog box, select **Cursors** mode from the **Mode** option box, or use the **Ctrl+X** accelerator key. A wire frame appears around the object. The dialog box changes to add the Probe controls. Select the probe you want to set by choosing the **Probe(s)** option box. This opens an options menu with a list of the available probes, from which you can select the desired probe.
4. Use the mouse to select a point or points to use as input to the tool connected to the **Probe** or **ProbeList** icon.

   To add a point, double-click on the *left* mouse button inside the wire-frame box. A small square box appears, marking the point.

   Note: The **Probe** tool allows only one point, while the **ProbeList** allows several.

   To move a point, select the point by pressing the left mouse button with the mouse pointer positioned on it. When the left mouse button is depressed on the point, the three projections (one for each axis) appear inside the wire-frame box as dots, and the values for the *x, y,* and *z* coordinates are displayed on the right side of the Image window menu bar, as illustrated in Figure 40 on page 86. You can move the point by dragging the selected point inside the box along the same direction as any of the axes. When you have moved the point to the desired area, release the left mouse button.

   Note that since perspective projection does not preserve parallel lines, the directions in the axes diagram do not necessarily correspond with the direction

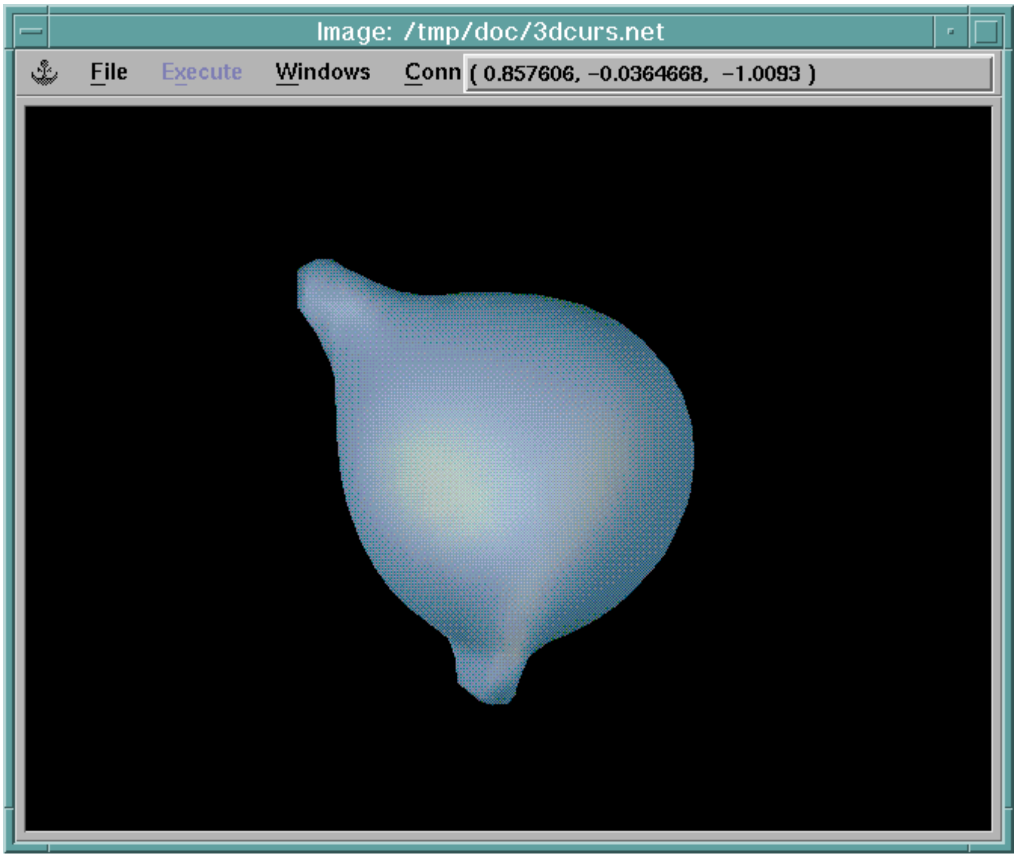


Figure 40. 3-D Cursor with a Selected Point

that the point moves in the Image window. However, these axes do correspond with the values of the coordinates.

While using orthographic projection, the movement of the point in the Image window corresponds to the directions of the axes diagram.

To delete a point, double click on it with the left mouse button.

You can restrict the movement of the 3-D cursor with the **Constraints** option. Selecting the **Constraints** option box reveals an options menu that lets you choose which of the three axis projections to that movement is constrained. For example, if the *x* axis is selected from the cascade menu, you are able to move only the *x* projection in the 3-D cursor box. When cursor movement is constrained, the portion of the wire-frame box that corresponds to the selected axis is highlighted. To remove movement constraints on the cursors, select **None** from the options menu.

Constraining is useful for more precise positioning of the cursor. Note that exact positioning is not possible with the 3-D cursor tool.

5. Repeat the previous three steps for each probe icon in your visual program.

   Reexecute the visual program to implement the probes.

While in **Cursors** mode, the center and right mouse buttons have the same functions as they do in **Rotate** mode.

Note: You can also enable Probe (cursors) mode by using the **intrctnMode** parameter to the Image tool (see Image in *IBM Visualization Data Explorer User's Reference*).

Using Pick

Picking consists of choosing a location on an object in an image using the mouse. A chosen location is called a "poke". Each poke may intersect the object in the image in one or more places (the intersections are called "picks") or may not intersect the object at all. For example, a poke on a spherical isosurface results in two "picks": one on the front of the sphere and one on the back. Picking differs from using probes, in which probes may be present anywhere in a 3-dimensional space, while picks always exist on the surface of an object.

After you execute a visual program and have an image in the Image window, you can modify the visual program to include a **Pick** tool from the **Special** category. The Pick tool accepts input from the mouse and outputs a field that specifies the picked point or points. The "positions" component of this field identifies each picked point on the object in the image. The field can be used, for example, to identify all picked points with a glyph, or to start streamlines at each picked point. In addition, the field output by the Pick tool can be used by a user-written module to perform a variety of operations on the object in the image (e.g., coloring each picked object a particular color). *IBM Visualization Data Explorer Programmer's Reference* includes a sample module that uses the pick structure in this way.

To use picking to select points on objects:

1. Execute a visual program to produce an image in the Image window.
2. In the VPE, place one or more pick tools from the **Special** category in the visual program, connecting them to the tools for which you want to provide input.

   The pick icons are numbered as you place them on the canvas. For example, the first pick icon you place is labeled "Pick_1", the second "Pick_2", and so on. You can change the label of the icon by using its Configuration dialog box.
3. In the **View Control...** dialog box, select **Pick** mode from the **Mode** option box or use the **Ctrl+I** accelerator key. The dialog box changes to add the Pick controls.

   Select the pick tool you want by choosing the **Pick(s)** option box. This opens an options menu with a list of the available picks from which to select.
4. Select a point or points as input to the tool connected to the Pick icon.

   To choose a point, click on a point in the image. A small square box appears, marking the point.

Depending on whether you have the **persistent** parameter to the Pick tool set to 0 or 1, subsequent executions may or may not use the last pick point or points chosen. If **persistent** is set to 0, then pick points are not saved between executions; if **persistent** is set to 1, then pick points are saved between executions.

Note: You can also enable Pick mode by using the **intrctnMode** parameter to the Image tool (see Image in *IBM Visualization Data Explorer User's Reference*).

Undo, Redo, and Reset

These options are almost self-explanatory. **Undo** and **Redo** both restore images, either by "undoing" the current image or by "redoing" the image that has just been "undone".

The **Reset** option returns the Image window to its initial state (i.e., before the image was first modified). See also "Restoring Images" on page 84.

AutoAxes...

This option generates an axes box around an image:

Select **AutoAxes...**from the **Options** pull-down menu. The **AutoAxes Configuration...** dialog box appears, as illustrated in Figure 41 on page 89.

The dialog box consists of eight parts:

Enabled The first "part" consists of a single toggle button—**AutoAxes enabled**. This button must be activated (depressed) to make an axes box appear the next time the visual program is executed. (The button is automatically activated whenever one or more of the AutoAxes options are changed.) Releasing (deactivating) the button prevents the appearances of an axes box.

Input groups The second part of the dialog box displays six (6) of the available configuration options shown below. Depressing (enabling) any of these option buttons automatically expands the window appropriately to reveal the relevant options. Releasing (disabling) any of these option buttons removes the relevant options.

Axes' Labels Allows you to individually specify a label for each of the three axes:

X, Y, Z

Miscellaneous Contains the following specifications:

Frame allows you to turn on or off a frame for the axes (that is, lines which complete the cube, in addition to the back three faces which are drawn by default).

Grid allows you to turn on or off grid lines along major ticks.

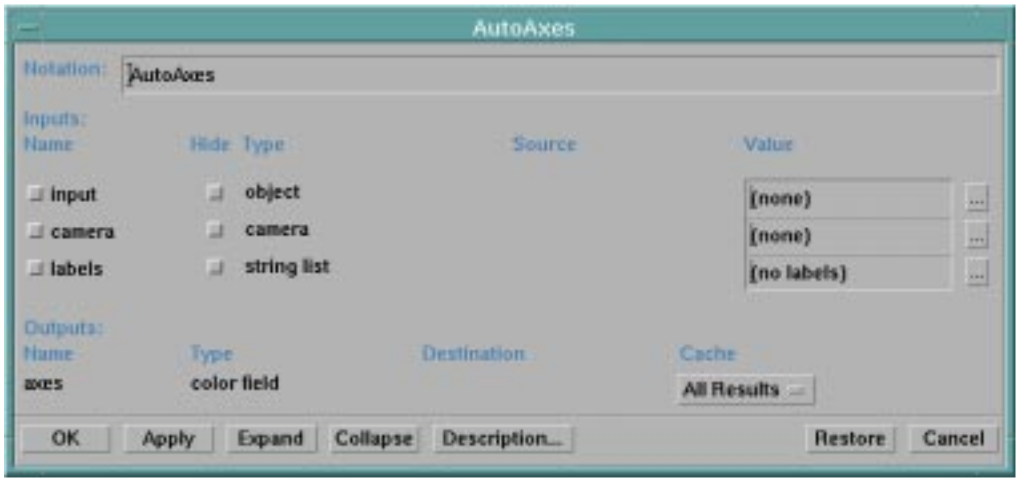
Font allows you to choose a font for the labels. The ellipses button allows you to choose from the set of predefined fonts.

Label Scale
allows you to change the size of the labels from the default size. For example, specifying a labels scale of 2 will make the labels twice as large.

Annotation Colors allows you to specify a color for each part of the axes: the grid (if drawn), ticks, labels, and background. The colors can be any of the defined colors (see "Color" on page 75 in *IBM Visualization Data Explorer User's Reference*), and in addition, the background can be drawn as "clear" (invisible).

Corners / Cursor The Corners section allows you to explicitly set the range of the axes in each dimension. The Cursor section allows you to place a cursor (marker) at a specific location in the axes box.

Ticks Allows you to specify number or locations of ticks. If the option menu to the right of "Ticks" is set to "All", then you can specify the approximate total number of ticks in the "All" field.


AutoAxes

Notation: AutoAxes

Inputs:

Name	Hide	Type	Source	Value
input		object		(none)
camera		camera		(none)
labels		string list		(no labels)

Outputs:

Name	Type	Destination	Cache
axes	color field		All Results

Figure 41. AutoAxes Configuration dialog box

If the option menu to the right of "Ticks" is set to "Per Axis", then you can specify the approximate number of ticks on each axis. If the option menu to the right of "Ticks" is set to "Values", then you can use the Ticks' Values section to set exact tick locations and labels. You can also specify the direction of the ticks to point inward or outward using the buttons at the bottom of this section

Ticks values This section is only enabled if the option menu to the right of Ticks is set to "Values" By using the ... buttons you can add Tick Location/Tick Label pairs for each axis. See "AutoAxes" on page 27 in *IBM Visualization Data Explorer User's Reference* for more information.

buttons **OK** and **Apply** both confirm option changes, which appear the next time the visual program is executed. **OK** also closes the dialog box.

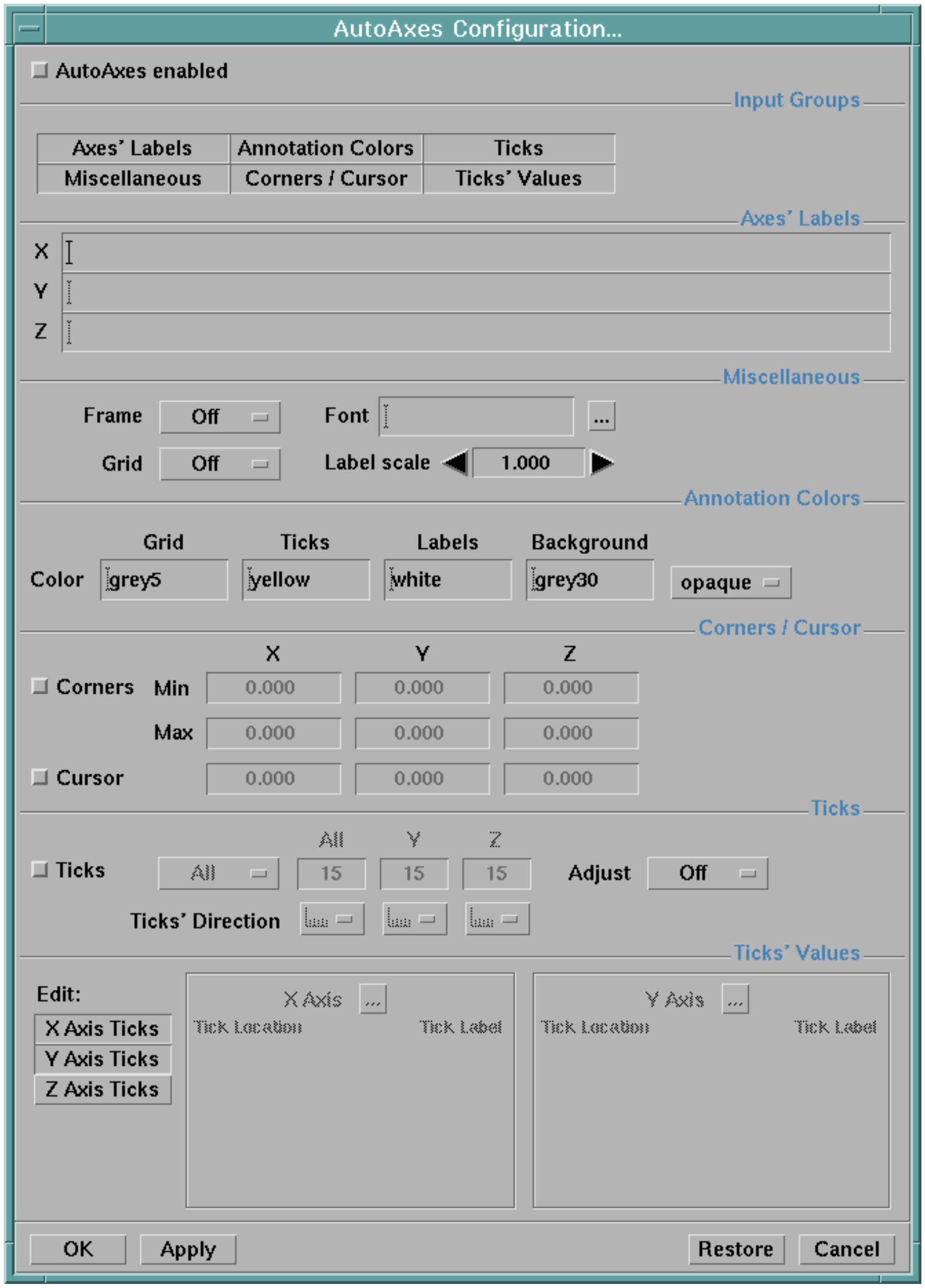
Restore and **Cancel** both restore values that were present when you opened the dialog box or last clicked on the **Apply** button. **Cancel** also closes the dialog box.

For more details on the AutoAxes configuration, see the corresponding module description in *IBM Visualization Data Explorer User's Reference*.

Note: It is also possible to set AutoAxes parameters using input parameters to the Image tool (see Image in *IBM Visualization Data Explorer User's Reference*).

Set Background Color...

This option displays a dialog box that will accept either a color-name string or an RGB vector as the specification of a background color for the Image window. Valid strings are listed in a user-specified lookup table or a file supplied with Data Explorer (see Color in *IBM Visualization Data Explorer User's Reference*). If neither is available, Data Explorer uses a smaller, internal list (see Appendix F, "Data Explorer Colors" on page 313).


AutoAxes Configuration...

☐ AutoAxes enabled

Input Groups

Axes' Labels	Annotation Colors	Ticks
Miscellaneous	Corners / Cursor	Ticks' Values

Axes' Labels

X

Y

Z

Miscellaneous

Frame: Off

Font:

Grid: Off

Label scale: 1.000

Annotation Colors

	Grid	Ticks	Labels	Background	
Color	grey5	yellow	white	grey30	opaque

Corners / Cursor

		X	Y	Z
☐ Corners	Min	0.000	0.000	0.000
	Max	0.000	0.000	0.000
☐ Cursor		0.000	0.000	0.000

Ticks

		All	Y	Z
☐ Ticks	All	15	15	15

Adjust: Off

Ticks' Direction

Ticks' Values

Edit:

X Axis Ticks

Y Axis Ticks

Z Axis Ticks

X Axis ...

Tick Location — Tick Label

Y Axis ...

Tick Location — Tick Label

OK — Apply — Restore — Cancel

Figure 42. Expanded AutoAxes Configuration Dialog Box. The box lists seven options not visible in the default version (Figure 41).

Notes:

1. Although each defined color has a corresponding RGB vector, the range of possible vectors is continuous from black ([0 0 0]) to white ([1 1 1]). Thus you can specify an RGB vector for which there is no corresponding string (i.e., no defined color). Nonetheless, Data Explorer will accept the vector as valid and generate the corresponding color.

2. Data Explorer supplies double quotation marks for color-name strings, and brackets, commas, and terminal zeros for vectors. For example, either `green` or `0 1 0` is sufficient to specify the color green.
3. It is also possible to set the background color using the **bkgrdColor** parameter to the Image tool (see Image in *IBM Visualization Data Explorer User's Reference*).

Display Rotation Globe

The rotation globe can be displayed only in the **Roam** and **Rotate** modes of the **View Control...** dialog box (see "Controlling the Image: View Control..." on page 74). In either mode, activation (depression) of the **Display Rotation Globe** toggle button generates a globe in the lower left-hand corner of the Image window. The orientation of this globe changes in parallel with that of the axes in the lower right-hand corner of the window, or vice versa, depending on which object you manipulate.

Hold down the left mouse button to rotate the objects in two dimensions (**Rotate** mode) and the middle button to orient them in three (**Roam** mode).

Rendering Options...

You can choose between software and hardware rendering if you are running Data Explorer on a workstation with a graphics card that supports hardware rendering. Approximations in both types of rendering help the user to see the effect of rotation, roam, or navigation interactions before rendering and display are complete.

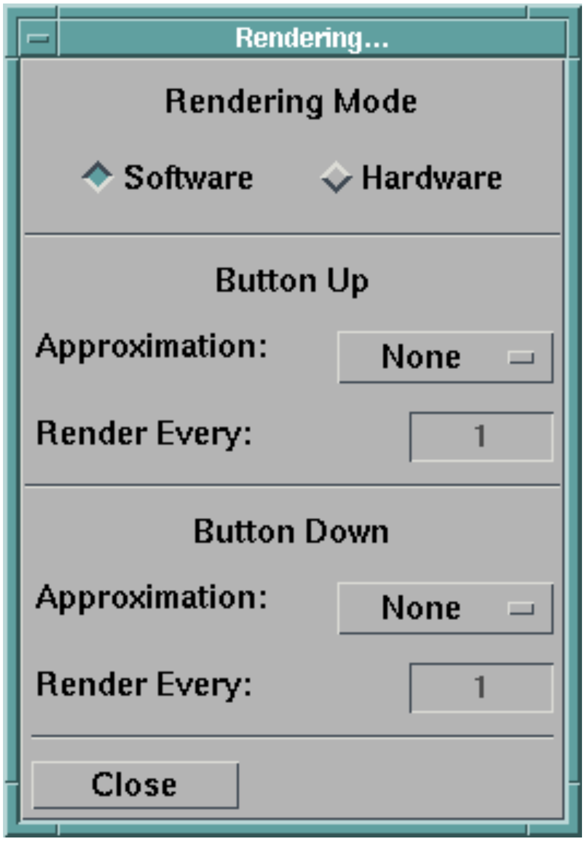
To set the rendering options, select **Rendering Options...** in the **Options** pull-down menu in the Image window. This causes the **Rendering...** dialog box to appear, as illustrated in Figure 43 on page 92.

The **Rendering...** dialog box allows you to select the rendering mode with a toggle button. If you select **hardware**, this means "use hardware if available" at the time Data Explorer is run. If a graphics card which supports hardware rendering is not available, software rendering will be used instead.

You can specify the approximation method that Data Explorer uses for each of two execution states—button-up execution and button-down execution. Button-down execution is applicable in execute-on-change mode and in navigation.

For example, you might specify **None** for the approximation method for button-up execution, and **Dots** for the approximation method for button-down execution. When holding the mouse button down in **Rotate** mode (with **Execute on Change** enabled), these rendering methods display a dot representation of the object, giving you quick feedback on your camera position, until you release the mouse button, at which point the object is rendered normally. The following is a brief description of the rendering methods you can specify with this dialog box. For detailed information about the rendering options, see "Display" on page 109 in *IBM Visualization Data Explorer User's Reference*.

None
No approximation method is specified, so a complete rendering of the image is done.


Rendering...

Rendering Mode

[✓ Software] Hardware

Button Up

Approximation: None

Render Every: 1

Button Down

Approximation: None

Render Every: 1

Close

Figure 43. Rendering Options Dialog Box

Wireframe (available only with hardware rendering)
: Renders the object as a wireframe, at the specified density. Surfaces are rendered as wireframe meshes, while points and volumes are rendered as dots. Wireframes are produced at full or fractional density.

Dots
: For hardware rendering, this approximation renders all points, surfaces, and volumes as dots, at the specified density. Lines are rendered as wireframe. Dots are produced at full or fractional density.

  For software rendering, all points, surfaces, and volumes are rendered as dots. You cannot specify the density for software rendering.

Box
: Draws the bounding box of each field in the object to be rendered.

While the Dots or Wireframe hardware rendering approximation methods are specified, the integer for **Render every** controls the density of the rendering approximation. For the Dots approximation, it means that every *n*th vertex is rendered, where *n* is the integer specified. For Wireframe approximation, the rendering depends on the type of connections. For example, if the connections are triangles, it means that every *n*th triangle is rendered. You can use the **Render every** value to control the speed of your rendering; the higher the value, the faster an object can be rendered. The default value is 1 (meaning every dot and every wireframe is rendered).

The **Render every** box is grayed out during software rendering, and during hardware rendering that is not Dots or Wireframe approximations.

Note: Do not use the Options module to set hardware-rendering options if you intend to use the **Rendering...** dialog box: options set by the Options module are overridden by those set through the dialog box (see "Display" on page 109 in *IBM Visualization Data Explorer User's Reference*.

Image Depth

This option determines the maximum number of possible colors (or shades of color) that Data Explorer can use in creating an image. With greater numbers, gradations are smoother and edges are sharper. The image is "clearer." The choices are the number of bits available for specifying colors: 8 (256 colors), 12 (4096) and 24 (more than 16 million). Data Explorer supports 8 bits. With the appropriate graphics card, it will process 12 and 24 bits.

Changing the Rate of Frame Display: Throttle...

You can specify a maximum rate of speed to display new frames. This rate is specified as a number of seconds per frame (i.e., the minimum number of seconds that any frame will be displayed before advancing to the next frame).

To change the rate of display, do the following:

1. Select the **Throttle...** option from the **Options** pull-down menu in the Image window. A dialog box appears, as illustrated in Figure 44 on page 94.
2. Edit the **Seconds per Frame** text field, changing it to the minimum number of seconds you want any given frame to be displayed.
3. Click on the **Close** button. The new rate takes effect.

You can also set the throttle value by using the **throttle** parameter to the Image tool (see Image in *IBM Visualization Data Explorer User's Reference*).

Changing the Title of an Image Window

By default, the title of the Image window is the name of the visual program that produced the image. If your visual program requires multiple image windows, it may be difficult to distinguish the windows, since all of the Image windows, by default, have the same title.

In this case, you can change the title of each Image window to help you organize them. To change the name of the Image:

1. Select the **Change Image Name...** option from the **Options** menu in the Image window. A dialog box opens.
2. Enter a new name for the image in the text field of the dialog box, and click on **OK**. The title bar of the Image window is updated to reflect the change.

Notes:

1. If you start Data Explorer with the Image window as the anchor without specifying a visual program in the command line, the title of the Image window defaults to "Image." However, when you load a visual program, the Image window assumes either the name of the visual program or, if the visual program contains any named images, the name of one of the Image windows.

2. It is also possible to specify the title of an Image window by using the **title** parameter to the Image tool (see Image in *IBM Visualization Data Explorer User's Reference*).

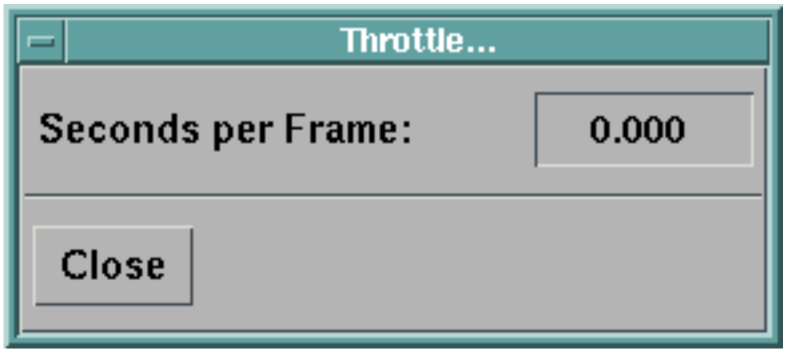


Figure 44. Throttle Dialog Box

Control Panel Access...

Clicking on this option in the **Options** pull-down menu generates a dialog box with two toggle buttons:

- The button on the left determines whether the specified control panel will be accessible from the Image window (using the **Open Control Panel by Name** option of the **Windows** pull-down menu) when Data Explorer is invoked with the `-image` or `-menubar` option.
- The button on the right (...), when activated (depressed) will display the control panel(s).

Saving an Image

Data Explorer provides you with the capability to save single images and image series to disk files. You save images using the **File** menu **Save Image** option of the Image window. Selecting this option causes a **Save Image** dialog box to appear (see Figure 45 on page 95.).

To save an image:

1. Open the Save Image dialog box using the **File** menu **Save Image** option.
2. Select the name of the file into which the image will be written or enter a file name into the **Output File Name** text field.
3. Select the image format.
4. Specify any image format options (e.g., **Gamma Correction**).
5. Click on **Save Current**.
6. Click on **Apply**.

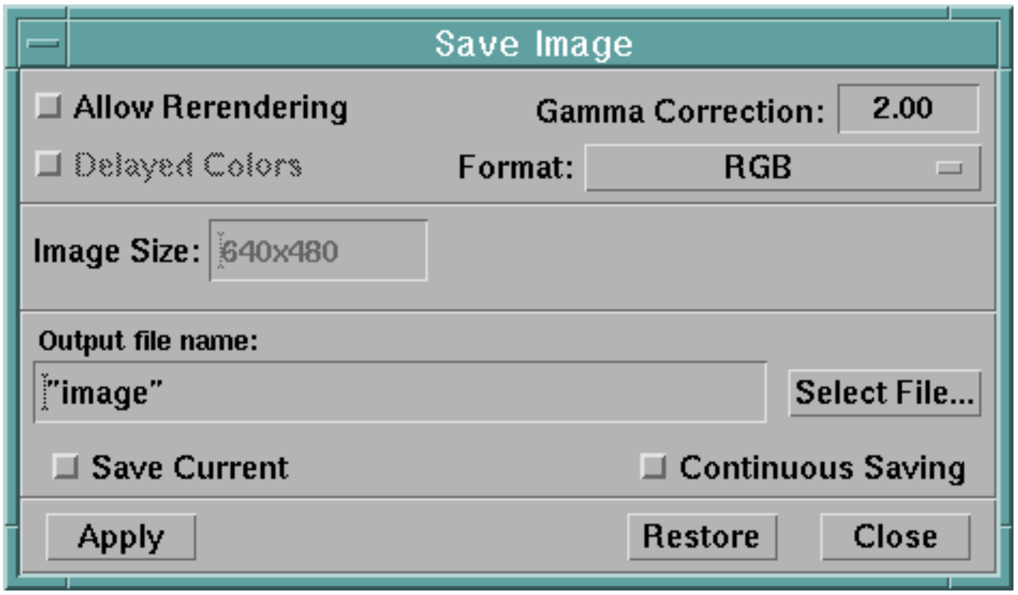
You can also save an image by using the **recordEnable**, **recordFile**, **recordFormat**, **recordRes**, **recordAspect** parameters to the Image tool. Portions of the **Save Image...** dialog box will be grayed out as appropriate if the corresponding parameter is set using these parameters to the Image tool. (See "Image" on page 160 in *IBM Visualization Data Explorer User's Reference*).

If you select (specify) an existing file and select one of the RGB, MIFF, or YUV formats, the image will be appended to the existing file. If you select TIFF or PostScript formats, the existing file will be overwritten by the new image.

When you want to save another image, you need only click on **Save Current**, specify a new file name, and click on **Apply**.

To save a continuous sequence of images

1. Open the Save Image dialog box using the **File** menu **Save Image** option.


Save Image

Allow Rerendering | Gamma Correction: 2.00

Delayed Colors | Format: RGB

Image Size: 640x480

Output file name: "image"

Save Current | Continuous Saving

Figure 45. Save Image Dialog Box

2. Select the name of the file into which the sequence will be written or enter a file name into the **Output File Name** text field.
3. Select the image format, either "RGB", "R+G+B" "MIFF", or "YUV".
4. Depress the **Continuous Saving** toggle by clicking on it.
5. If you do not wish to include the currently displayed image in the sequence, be sure the **Save Current** toggle button is not depressed.
6. Click on the **Apply** button.

Each time an image is displayed in the Image window it will be appended to the file specified in the Selection text field. To disable the continuous saving mode:

1. Open the Save Image dialog box using the **File** menu **Save Image** option.
2. Release the **Continuous Saving** toggle button by clicking on it.
3. Click on the **Apply** button.

Save Image Options

Allow Rerendering allows you to specify whether or not the currently displayed image should be rerendered at a new resolution or aspect ratio.

If **Allow Rerendering** is toggled off, then the **Image Size** is completely determined by the resolution of the currently displayed image. **Image Size** and **Output PPI** cannot be independently controlled, and you can not change the aspect ratio of the image from that of the currently displayed image. If you attempt to change the aspect ratio of the image size (e.g. by specifying a size of 8x8 when the original image is not at an aspect ratio of 1:1), the new image size will not be accepted by the **Image Size** text field. If you change only one component of the image size (e.g. by specifying "8 x"), then the other component of the image size will be computed by comparing it to the current aspect ratio, and **Output PPI** will be adjusted such that the number of pixels of the image remains that of the currently displayed image.

If **Allow Rerendering** is toggled on, then the currently displayed image will be rerendered at a new resolution based on the settings of **Output PPI** and **Image Size**, which can now be independently controlled. You can use this option, for example, to save or print an image at a much higher resolution than is displayed on the screen. You can also specify the number of pixels directly by setting the units of **Image Size** to pixels (note that in this case it is meaningless to set the Output PPI).

Gamma Correction	Allows you to specify gamma correction applied to the output image. The default is 2.
Delayed Colors	For TIFF, MIFF, and PostScript image formats, allows you to specify whether the image is saved in an image-with-colormap format. For GIF image format, this option is required by the format.
Format	Allows you to choose from the set of available image formats.
Output file name	Allows you to specify the file name to which the image should be written. An appropriate extension for the chosen format will be added if you do not provide one.
Select File	Allows you to use a File Selection dialog to specify the output image file name.
Save Current	Allows you to specify that the current image should be saved when the **Apply** button is pressed.
Continuous Saving	Allows you to save a series of images. If you do not want the current image to be saved, be sure the **Save Current** button is not set before pressing **Apply**. When **Continuous Saving** is activated, each image displayed in the image window will be saved to the specified file. This option is useful only for image formats which support series: RGB, R+G+B, YUV, and MIFF.

For all formats other than the Postscript formats, the following field is available:

Image Size	If the **Allow Rerendering** button is set, this field allows you to set the resolution of the output image to something other than that of the image displayed to the screen.

If one of the Postscript formats is selected then the following fields are displayed:

Image Dimensions	Allows you to specify the size of the image on the page (by default in inches).
Orientation	Allows you to specify the orientation of the image to portrait or landscape, or automatic, which chooses the best for the given image.
Input Image Size	Specifies the resolution of the image to be saved. This field is enabled only if the **Allow Rerendering** button is toggled on.
Page Dimensions	Specifies the size of the page. By default this is specified in inches.

Output PPI Specifies the “pixels per inch” of the output image.

Note: Unless you specifically care to set the precise pixels per inch, you do not typically need to set this.

Margin Width Specifies a margin width of white space on the page.

For PostScript formats, the printed image will, by default, fill the page to within **Margin Width** of the edge of the page. If **Allow Rerendering** is off, the pixels in the image will be sized appropriately to scale the image to fill the page, but the same number of pixels as in the currently displayed image will be used. If this results in a grainy image, set **Allow Rerendering** on, and enter a different **Input image size**. For example, if the displayed image is 640x480, and you want to double the resolution, just enter 1280 in the **Input image size** field and Data Explorer will recalculate the new value of y (960) and the new (higher) value for **Output PPI**.

By default, **Image Dimensions**, **Page Dimensions**, and **Margin Width** are specified in inches However, you can use the `DX*metric` resource or the `-metric` command line option to use centimeters instead. See Table 7 on page 299.

Pushbuttons **Apply** causes the currently displayed image to be saved if the **Save Current** toggle button is depressed.

Restore restores settings in the dialog to what they were the last time the **Apply** button was depressed.

Close causes the dialog to be closed without saving an image.

Note: If you are not using the Image window, this functionality is available with the WriteImage module. See “WriteImage” on page 374 in *IBM Visualization Data Explorer User’s Reference*.

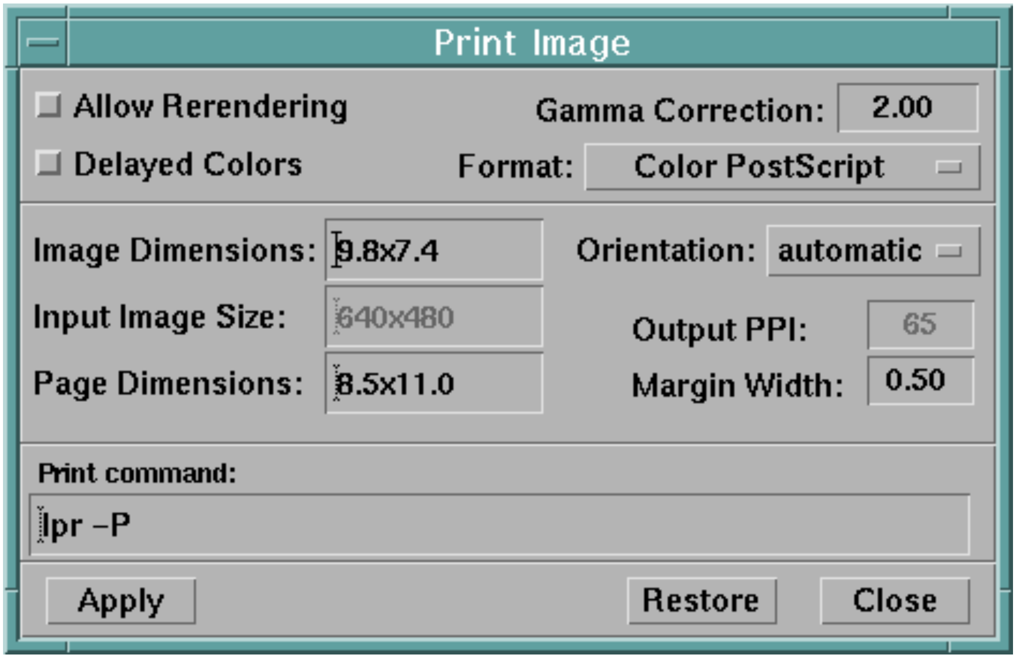
Printing an Image

You can print images displayed in the Image window by choosing the **Print Image...** option from the **File** menu of the Image window. Selecting this option causes a Print Image dialog box to be opened (see Figure 46 on page 98). Note that portions of the **Print Image** dialog box will be grayed out as appropriate if the corresponding parameter is set using the **recordFormat**, **recordRes**, or **recordAspect** parameters to the Image tool.

Print Image Options

Allow Rerendering allows you to specify whether or not the currently displayed image should be rerendered at a new resolution or aspect ratio.

If **Allow Rerendering** is toggled off, then the **Image Size** is completely determined by the resolution of the currently displayed image. **Image Size** and **Output PPI** cannot be independently controlled, and you can not change the aspect ratio of the image from that of the currently displayed image. If you attempt to change the aspect ratio of the image size (e.g. by specifying a size of 8x8 when the original image is not at an aspect ratio of 1:1), the new image size will not be accepted by the **Image Size** text field. If you change only one component of the image size (e.g. by specifying “8 x”), then the other component of the image size will be computed by


Print Image

Allow Rerendering

Delayed Colors

Gamma Correction: 2.00

Format: Color PostScript

Image Dimensions: 9.8x7.4

Orientation: automatic

Input Image Size: 640x480

Output PPI: 65

Page Dimensions: 8.5x11.0

Margin Width: 0.50

Print command:

lpr -P

Figure 46. Print Image Dialog Box

comparing it to the current aspect ratio, and **Output PPI** will be adjusted such that the number of pixels of the image remains that of the currently displayed image.

If **Allow Rerendering** is toggled on, then the currently displayed image will be rerendered at a new resolution based on the settings of **Output PPI** and **Image Size**, which can now be independently controlled. You can use this option, for example, to save or print an image at a much higher resolution than is displayed on the screen. You can also specify the number of pixels directly by setting the units of **Image Size** to pixels (note that in this case it is meaningless to set the Output PPI).

Gamma Correction	Allows you to specify gamma correction applied to the output image. The default is 2.
Delayed Colors	For TIFF, MIFF, and PostScript image formats, allows you to specify whether the image is saved in an image-with-colormap format. For GIF image format, this option is required by the format.
Format	Allows you to choose from the set of available image formats.

For all formats other than the Postscript formats, the following field is available:

Image Size	If the **Allow Rerendering** button is set, this field allows you to set the resolution of the output image to something other than that of the image displayed to the screen.

If one of the Postscript formats is selected then the following fields are displayed:

Image Dimensions	Allows you to specify the size of the image on the page (by default in inches).

Orientation	Allows you to specify the orientation of the image to portrait or landscape, or automatic, which chooses the best for the given image.
Input Image Size	Specifies the resolution of the image to be printed. This field is enabled only if the **Allow Rerendering** button is toggled on.
Page Dimensions	Specifies the size of the page. By default this is specified in inches.
Output PPI	Specifies the "pixels per inch" of the output image. **Note:** Unless you specifically care to set the precise pixels per inch, you do not typically need to set this.
Margin Width	Specifies a margin width of white space on the page.

For PostScript formats, the printed image will, by default, fill the page to within **Margin Width** of the edge of the page. If **Allow Rerendering** is off, the pixels in the image will be sized appropriately to scale the image to fill the page, but the same number of pixels as in the currently displayed image will be used. If this results in a grainy image, set **Allow Rerendering** on, and enter a different **Input image size**. For example, if the displayed image is 640x480, and you want to double the resolution, just enter 1280 in the **Input image size** field and Data Explorer will recalculate the new value of y (960) and the new (higher) value for **Output PPI**.

Print command Contains a text field where you can enter a command to print the image, for example:

```
lpr -P myPrinter
```

Pushbuttons

Apply Causes the command specified by **Print Command** to be executed.

Restore Restores settings in the dialog to what they were the last time that **Apply** was depressed.

Close Causes the dialog to be closed without printing an image.

6.2 Using the VPE

A visual program is a collection of interconnected tools that acts upon one or more inputs to create one or more outputs (for example, an image). You derive program inputs from the output of tools in the program, or by setting the input values to constants. Tools that provide output values as input to other tools are (in alphabetical order):

- Colormap Editors (see 6.3, "Using the Colormap Editor" on page 119)
- Interactors (see "Using Interactors" on page 142)
- Macros (see 7.2, "Creating and Using Macros" on page 149)
- Modules (see Chapter 1, "Data Explorer Tools" on page 1 in *IBM Visualization Data Explorer User's Reference*)
- Picks (see "Using Pick" on page 87)
- Probes (see "Using Probes (Cursors)" on page 85)
- Sequencers (see "Using the Sequencer" on page 68)
- Transmitters and Receivers (see "Using Transmitters and Receivers" on page 106).

Before building a visual program, you should be familiar with the information presented in this chapter as well as the information on Data Explorer modules presented in Chapter 1, “Data Explorer Tools” on page 1 in *IBM Visualization Data Explorer Programmer’s Reference*.

Chapter 2, “Tutorial I: Using Data Explorer” on page 3 in *IBM Visualization Data Explorer QuickStart Guide* introduces the Data Explorer graphical user interface. Chapter 3, “Tutorial II: Editing and Creating Visual Programs” on page 21 in *IBM Visualization Data Explorer QuickStart Guide* introduces the basic aspects of working with visual programs. These tutorials will familiarize you with the user interface as well as some of the commonly used Data Explorer modules.

Creating a Visual Program

To create a visual program, use the Visual Program Editor (VPE) window, in which you place and connect tools and specify values for those tools. Figure 47 illustrates the VPE window and a sample visual program.

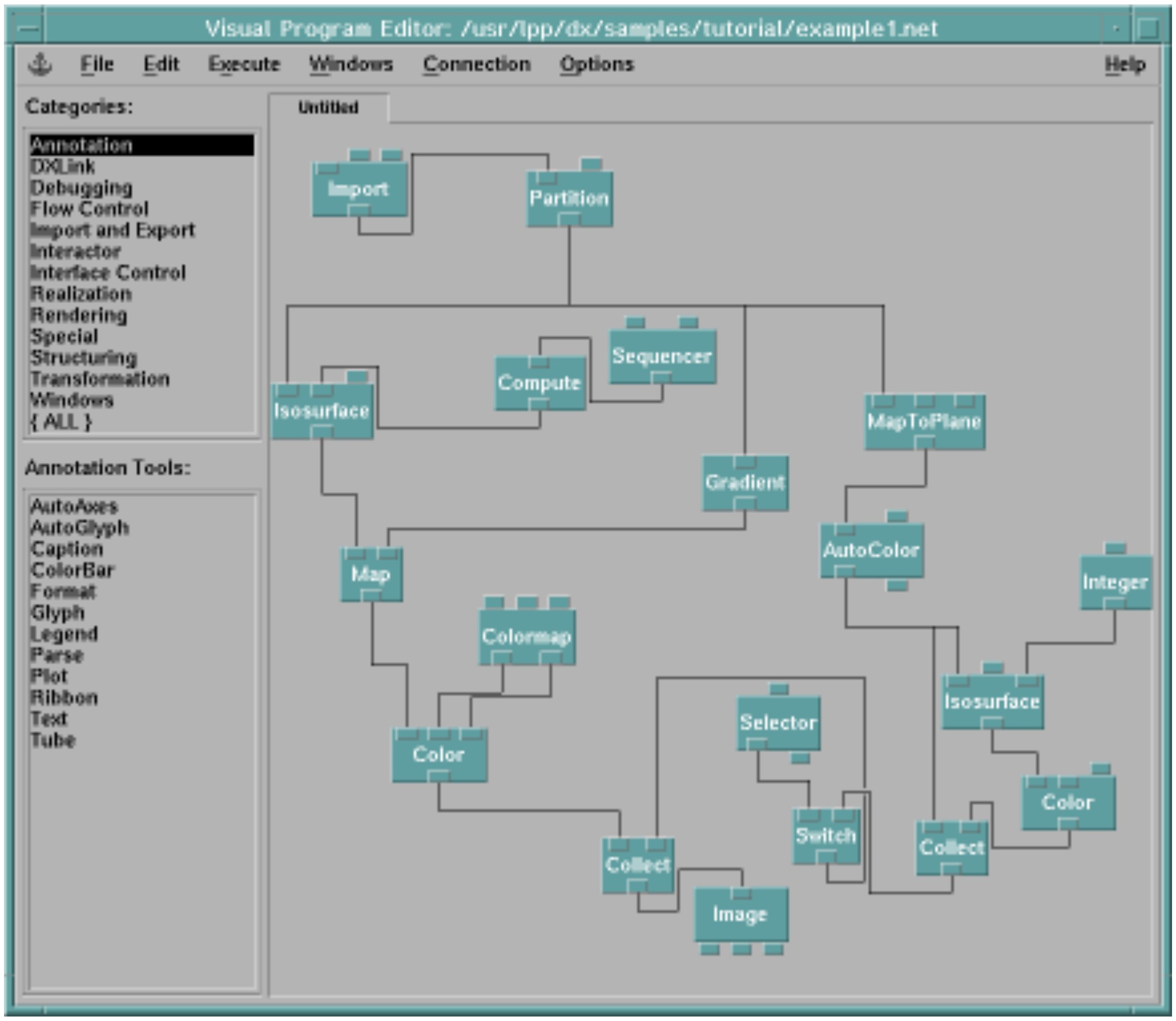


Figure 47. VPE Window

The title bar of the VPE displays the name of the current visual program file. On the left side of the window are two palettes. The top palette contains tool categories. When you select a category from the top palette, the bottom palette displays the tool names in that category. The large area on the right side of the window is called the *canvas*. You place tools on the canvas to construct a visual program.

In some cases, the visual program may be too large to be displayed all at once on the canvas. On the right and bottom sides of the canvas are *scroll bars*, which allow you to move the display to different parts of the visual program. To display a different part of the visual program, click on the arrows at the end of the scroll bars, or click and hold the button on the bar between the arrows and move it until the desired portion of the visual program is displayed. Scroll bars may also appear in the tool palettes, if the list of categories or tools are too long to fit.

When you place a tool onto the canvas, it is represented by an icon. Figure 48 illustrates an example of a *tool icon*.

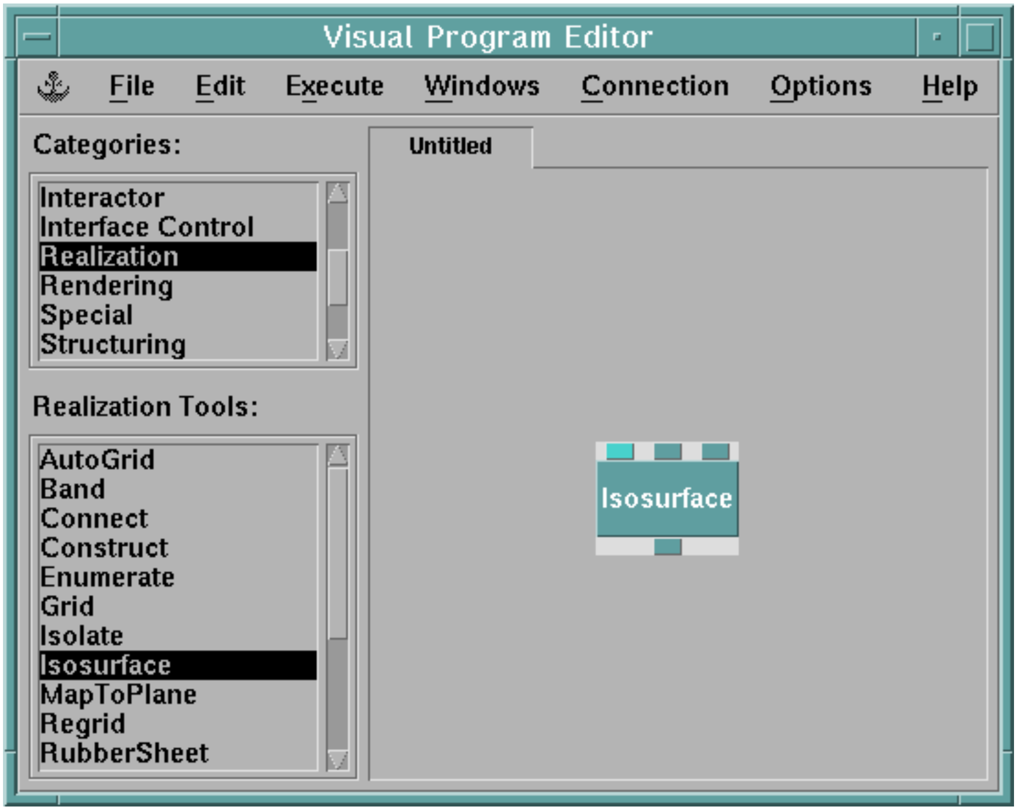


Figure 48. Example of a Tool Icon

The tool icon, which has a highlighted border when selected, consists of a rectangle displaying the name of the tool, plus one or more tabs on the top, the bottom, or the top and the bottom of the rectangle. The rectangle is the active area and is used for selecting and moving the tool. The tabs on the top represent inputs to the tool, and the tabs on the bottom represent outputs. It is possible for a tool to have either no inputs or no outputs. Some of a tool's input tabs may be hidden (i.e., not displayed). It is possible to reveal the hidden tabs or hide additional tabs. It is also possible to add or remove input tabs for tools that allow a variable number of inputs. On many tools, one or more of the inputs are highlighted with a different color, indicating that the input is required. These tabs are discussed in more detail in “Specifying Values for a Tool's Inputs” on page 103.

The basic steps in creating a visual program are:

1. Select and place the desired tools on the canvas.
2. Connect tool outputs to inputs.
3. Set values for the tools.
4. If you plan to change input parameters frequently while viewing an image (e.g., an isosurface value), then you should build a Control Panel and set interactors (see "Building Control Panels" on page 129 and "Using Interactors" on page 142).

Placing Tools on the Canvas

The tools are divided into categories, and a list of these categories appears in the top palette on the left side of the VPE window. To locate a tool:

1. Click on a desired category. The names of the tools in that category appear in the lower tool palette. All palette lists are presented in alphabetic order.
2. Select the appropriate tool name from the bottom palette by clicking on the tool's name. Clicking on an already selected tool deselects that tool.

To place one instance of the selected tool on the canvas:

1. Move the cursor to where you want to position the tool on the canvas. Note that the style of the cursor changes when you move it onto the canvas. Exact placement of the tool depends on the grid settings (see "Customizing the VPE Window" on page 113).
2. Click the mouse. The tool icon appears at the specified location. When you have placed the tool, its name is no longer highlighted in the lower palette.

To place multiple instances of the tool:

1. Double click on the tool name in the lower tool palette.
2. Click the mouse on the canvas to place one instance of the tool. Repeat this step for further instances of the tool.
3. To stop placing the same tool, deselect the tool's name in the palette by clicking on any tool name.

Note: After you place a tool, its icon stays selected until you place another tool, or deselect the tool as described in "Selecting, Moving, and Deleting Tool Icons".

If you place one tool icon on top of another, the bottom tool icon is pushed to the right to allow room for the new one. To avoid the displacement of tool icons, allow enough space for each tool icon you plan to use.

To deselect a tool name in the lower palette, do one of the following:

- Click on the tool name.
- Select another tool name.
- Select another category.

Selecting, Moving, and Deleting Tool Icons

To select a tool icon, click on it.

To select a group of tool icons, use one of these methods:

- Hold down the Shift key and click on each tool icon in turn.
- Position the cursor on the canvas near a corner of the group and drag the mouse to draw a selection box around the tool icons you want. To select the tool icon, you must completely enclose it in the selection box. As a tool icon is

encompassed by the selection box, it is highlighted (indicating that it is selected).

Note: If you have tools selected already and you want to select more, hold down the Shift key and either click on a single icon, or drag the selection box over several icons to select them. To select all icons on the canvas, use the **Select All** option of the **Edit** pull-down menu.

To deselect an icon, shift-click on it.

To deselect a group of tools, use one of these methods:

- Shift-click on each icon.
- Shift-drag to draw a box around the tools you want to deselect. As a selected tool is encompassed in the box, it becomes unhighlighted. Release the button to deselect the tools.

Note: Clicking on an empty part of the canvas deselects all selected tools.

To move a tool icon:

1. Press and hold the left mouse button on the tool icon and drag it to the desired location. While you are dragging the tool, an outline of the tool icon follows the motion of the mouse, but the tool icon remains in the original location.
2. Release the mouse button at the desired location. The tool icon moves to that location. Any lines connecting to other tools are rerouted to the tool's new location.

To move a group of tool icons:

1. Select a group of tool icons to be moved.
2. Position the cursor on any member of the group and drag it. An outline of each tool icon follows the motion of the mouse.
3. Release the mouse button. The icons representing the tools move to the new location. Any affected connections are rerouted.

To delete tool icons:

1. Select the tool icon or group to be deleted.
2. Press the **Ctrl+Delete** accelerator key or click on the **Delete** option of the **Edit** menu.

Specifying Values for a Tool's Inputs

To provide the values of a given input for a tool, use one of these methods:

- Connect the output tab of another tool to the desired input tab.
- *or*
- Use the tool's Configuration dialog box.

When you specify an input value for a tool, the corresponding tab on the tool's icon automatically folds in. If you do not specify an input value, the input tab remains folded out, indicating that the input is unbound and the tool uses its default for that input. The default values are given in the Configuration dialog box. The tabs that are folded in give you a visual representation of the inputs that have user-specified values. Some tabs require an input; those tabs remain highlighted in a different color until you specify an input with a connection or through a Configuration dialog box. Some of the tabs are not displayed, termed hidden. The hidden tabs can be revealed or additional tabs can be hidden (see “Revealing and Hiding Input Tabs” on page 110). Also, some tools allow you to add or remove input tabs (see “Adding and Removing Input and Output Tabs” on page 106). Figure 49 on page 104 illustrates tools whose inputs have been specified.

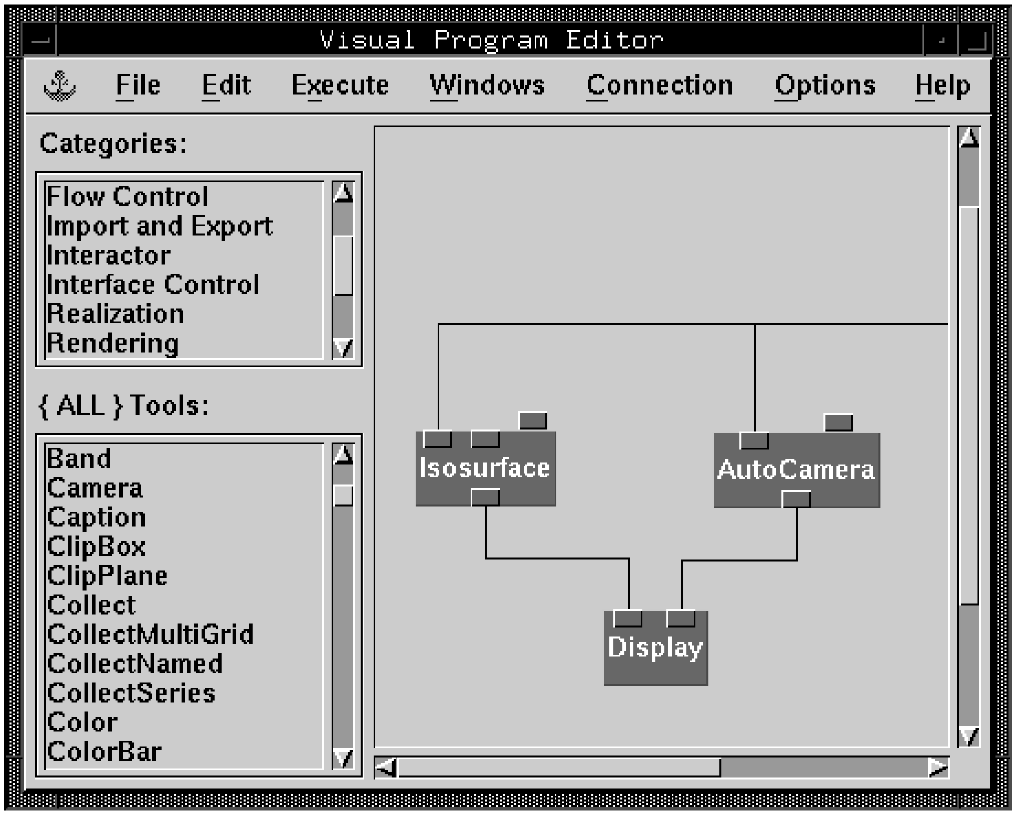


Figure 49. How Tabs Work. An input tab folds in when its input is supplied through a connecting “arc” or is defined in the module’s configuration dialog box. When the default input is used, the tab remains up. An output tab folds in when it is connected to another module by an arc.

A tab that is folded in but has no connecting line leading to it indicates that its input was specified using the tool’s Configuration dialog box.

Creating, Deleting, and Moving Tab Connections

To connect tabs:

1. Click and hold the mouse button on the output tab. This causes the cursor to change to a downward pointing arrow. When the cursor is placed over the output tab, the name of the tab is displayed in the tool icon.
2. While pressing the mouse button, drag the cursor to the desired input tab. While you are dragging the cursor from the output tab, a white line appears indicating that a connection is being created.
3. When you enter the area encompassing a tool icon, the tabs that are compatible with the output tab you are trying to connect will change color. When the cursor is placed over an input tab, the name of the tab is displayed in the tool icon.

Note: You can also connect tabs by starting at the input tab. Simply press and hold the mouse button on the desired input tab; this causes the cursor to change to an upward pointing arrow. Drag the cursor to the desired output tab. When the cursor enters the area encompassing a tool icon, the output tabs that are compatible with the input tab will change color.

4. Place the pointer over the input tab you want and release the mouse button. If you release the mouse button in the rectangle area of the icon, the line automatically connects to the leftmost of the newly colored tabs.

   When you release the mouse button, the white temporary line is replaced with a black line, and the corresponding tabs fold in.

   Note: Sometimes, when tools are placed too close together on the canvas, a connection that you made is not visible. In this case, the affected input tabs are still folded in, and a dark line is visible on each one, but the connecting line is not visible. To correct this, move one of the affected tools to another spot on the canvas.

You can connect an output tab to more than one input tab (either on different tool icons or on the same tool icon). Therefore, you can make a connection from an output tab even though the output tab is folded in.

An input tab can receive a value from only one source, either from a connection to an output tab or from a value entered in a Configuration dialog box. Once an input tab folds in, you cannot make a new connection to the tab until the tab is released.

Because output tabs can have multiple destinations, connections can only be moved or deleted from an input tab.

To delete a tab connection:

1. Depress the mouse button on the input tab.
2. Drag the cursor to an empty space on the canvas (away from the active area of the tool). A white line from the output tab follows the mouse pointer. If you decide at this point *not* to break the connection, you must place the cursor back on the input tab where the connection was, and release the mouse button.
3. Release the mouse button. The connection is deleted and the input tab released.

To move a tab connection:

1. Depress the mouse button on the input tab.
2. Drag the cursor to an empty input tab (one that has not already been folded in).
3. Release the mouse button. The connection is rerouted to the new input tab.

Moving and Copying Tools

The Drag and Drop capability provided by Data Explorer allows you to move or copy selected tools within a Data Explorer window (e.g., the VPE or a control panel) or between different Data Explorer windows (i.e., between two different VPE windows). You can also drag interactor stand-ins to control panels to create interactors.

To initiate a drag and drop, first select the tools you want to move or copy, either by rubber-band or shift selection. Then place the mouse cursor over one of the selected tools and press the middle mouse button. Keeping the button pressed, move the cursor to an empty spot on the canvas and release the button. A copy of the selected tool will be placed where the mouse button was released. To do a cut and drop (move), use Shift-middle mouse button.

When dragging and dropping tools from the VPE, you may drop the tools on a VPE in another Data Explorer session or in another VPE (i.e., macro editor) of the same Data Explorer session. Interactors in control panels can only be dragged and dropped between panels in the same visual program.

If you drag a set of tools to an inappropriate window (e.g., from a control panel to a VPE), no action will occur. If you drag a set of tools that includes transmitters, the latter (along with their corresponding receivers) will be renamed to wireless-1, wireless-2, and so on.

Note: To include an existing `.net` file in the VPE, use the **Insert Visual Program** option of the **Edit** pull-down menu in the VPE. The inserted visual program is placed to the right of the rightmost tool in the existing network.

Using Transmitters and Receivers

It is possible to create large visual programs in Data Explorer. Data Explorer provides two tools, Transmitter and Receiver, to maintain the modularity and readability of large programs. These tools allow connections between input and output tabs without using a visible connecting line. Using the Transmitter and Receiver tools allows you to separate visual programs into logical blocks. For example, the output of several logical blocks can be transmitted to another block that receives them, collects them, and produces the image. Macros, described in 7.2, “Creating and Using Macros” on page 149, provide another way to structure visual programs into logical blocks.

To remotely connect input and output tabs:

1. Select the Transmitter tool, which appears under the Special category in the tool palette, and place it near the output tab.
2. Connect the tool’s output tab to the input tab of the Transmitter.
3. Select the Receiver tool (also in the Special category), and place it near the input tab to be connected to the output tab above.
4. Connect the Receiver’s output tab to the receiving tool icon’s input tab.

   The Receiver automatically assumes the same name as the Transmitter. There can be multiple instances of a Receiver corresponding to a single Transmitter. These Receivers assume the same name until a new Transmitter is selected.

   The name of a Transmitter and Receiver can be changed using the notation field of the Configuration dialog box (as described in “Entering Values in a Configuration Dialog Box” on page 107). When you change the name of the Transmitter, all Receivers that share a name with that Transmitter also change their names. However, when you change the name of a particular Receiver, the associated Transmitter and the other Receivers are not affected.

Adding and Removing Input and Output Tabs

Most of the tools have a fixed number of inputs and outputs, but some, such as **Collect** and **Compute**, allow the number of inputs to vary. For example, the default number of inputs for **Compute** is two, but you may want to use the output from an expression that has six inputs. Data Explorer lets you change such tools to accommodate the extra inputs.

To add input tabs to a tool on the canvas:

1. Select the tool icon by clicking on it.
2. Select the **Add Input Tab** option from the **Input/Output Tabs** option from the **Edit** pull-down menu. (You can also use the **Ctrl+A** accelerator key.)

   Note: If the tool you selected has a fixed number of inputs, the **Add Input Tab** option is grayed out.

The appropriate number of input tabs are added to the tool icon. Typically, the **Add Input Tab** option adds one tab to the icon. In the case of some tools, such as CollectNamed, that require inputs in pairs, two input tabs are added. To add multiple tabs, repeat the previous steps.

When you change the number of input tabs on the tool icon, the tool's Configuration dialog box is updated to reflect the change.

To remove input tabs from a tool on the canvas:

1. Select the tool by clicking on it.
2. Select the **Remove Input Tab** option from the **Input/Output Tabs** option from the **Edit** pull-down menu. (You can also use the **Ctrl+R** accelerator key.)

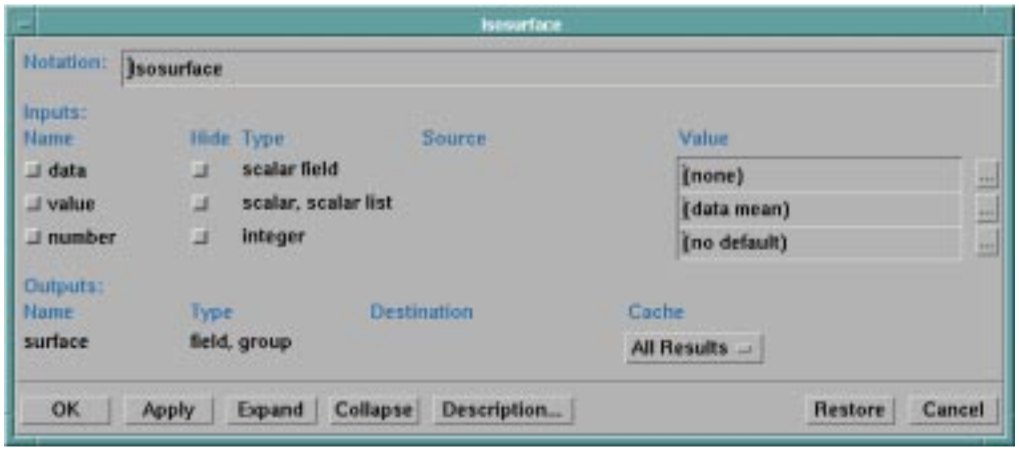
   Note: If the tool you selected has a fixed number of inputs, or if the icon has only the minimum number of tabs required for that tool, the **Remove Input Tab** option is grayed out.

   The appropriate number of input tabs are removed from the right-side tool icon. If the tabs that are removed previously had connections to them, those connections are broken. It is possible for some tools to have zero input tabs. For example, you may want the **Compute** module to contain only an expression, with no inputs. The output would be the result of the expression.

Entering Values in a Configuration Dialog Box

The Configuration dialog box displays the current state of a tool's inputs. You can modify the contents of the box directly. The contents are also automatically updated when changes in the visual program or in interactors in a Control Panel affect a tool's inputs. In general, specifying input values using the Configuration dialog box should be reserved for those values that remain constant during a visualization session. Use an interactor set in a Control Panel to specify values that are likely to be changed frequently.

This section describes a typical Configuration dialog box, illustrated in Figure 50 on page 108. The Configuration dialog box for the Compute module, which is unlike the other Configuration dialog boxes, is described in "Using the Compute Module Configuration Dialog Box" on page 111.



Notation: Isosurface

Inputs:

Name	Hide	Type	Source	Value
data		scalar field		(none)
value		scalar, scalar list		(data mean)
number		integer		(no default)

Outputs:

Name	Type	Destination	Cache
surface	field, group		All Results

Figure 50. Typical Configuration Dialog Box

You can open a Configuration dialog box in one of the following ways:

- Double-click on the rectangular portion of a tool's icon.
- Select the icon and click on the **Edit** menu **Configuration** option (or use the **Ctrl+F** accelerator key).

Note: If the tool icon is an interactor stand-in, a colormap editor stand-in, or an image tool, you must use the **Edit** menu **Configuration** option (or use the **Ctrl+F** accelerator key) to open the Configuration dialog box.

A typical Configuration dialog box consists of four major parts:

- Notation
- Inputs section
- Outputs section
- Pushbuttons

The following sections describe the elements in a Configuration dialog box.

Notation Field

By default, the Notation field displays the name of the tool. You can use this field to enter a short notation about the use of the tool in the current visual program:

1. Select the field by clicking on it.
2. Edit the field as a normal text field (i.e., using Backspace, Delete, and the alphanumeric keys).

In the case of Transmitter, Receiver, Probe, ProbeList, and Pick tools, use the Notation field to rename the tool, thus changing the name appearing on the tool icon.

Inputs Section

Toggle buttons

Specifies whether an input is active; that is, whether the input tab on the icon is folded in. The toggle buttons are the small square buttons on the far left of the dialog box.

The toggle buttons provide a visual indication of the inputs that have been specified. These buttons are analogous to the input tabs on the tool icon and are coherent with the tabs: the buttons can be activated and deactivated by clicking on them.

- When you activate a toggle button, the value of the input is either the output of another module (specified in the Source field of the dialog box), or the value specified in the Value field.
- When you deactivate a toggle button, the input parameter is unbound and the tool uses its default value for the input.

Name
Specifies the name of an input parameter. You cannot modify this field. This field is grayed out if the name is specified with tab connections.

Hide
Indicates whether an input tab is to be hidden. When you activate the **Hide** toggle button, the corresponding tab on the tool icon is removed. If a tab is connected to another tool it cannot be hidden. Once a tab is hidden, it can be removed from display in the Configuration dialog box by using the **Collapse** button.

To reveal individual input tabs, first click on the **Expand** button. This will cause the Configuration dialog box to resize, displaying all of the tool's inputs. Then deactivate the **Hide** toggle buttons of the desired inputs by clicking on the toggle buttons. As the inputs are revealed, the tool icon is updated to reflect the additional inputs. Once you have revealed all the desired tabs, you can click on the **Collapse** button to remove the remaining hidden tabs from being displayed in the Configuration dialog box.

See also "Revealing and Hiding Input Tabs" on page 110.

Type
Specifies the type of an input parameter. You cannot modify this field.

Source
Displays the name of the tool connected to the input, if a connection exists. If the name of a tool is displayed in the source field, the toggle button is activated and the name field is grayed out, and this input cannot be modified until the connecting line is deleted.

Value
You use the Value field to specify a value for an input. Initially the field contains the default value for an input. To modify the Value field, select the field by clicking on it. Then, edit it as a normal text field. The value must be specified in the syntax described in Chapter 10, "Data Explorer Scripting Language" on page 187. After you change the field, do one of the following:

- Press Enter (which automatically activates the toggle button if it is not already activated).
- Activate the toggle button by clicking on it.
- Click on either the **OK** pushbutton or the **Apply** button at the bottom of the box (see button descriptions in "Pushbuttons" on page 110).

Data Explorer automatically adds the appropriate delimiters for the type of value entered. For instance, if you specified a string parameter in the Configuration dialog box, Data Explorer automatically adds the quotes around it. If you specified a scalar list, Data Explorer adds braces.

Note: If a source is displayed, you cannot modify the Value field. If you have modified the Value field and you want to use the default value, release the toggle button. With the toggle button, you can flip between the modified value and the default value.

...

This button, when enabled, brings up a list of possible values for the parameter. This list is for convenience only; you may enter values other than those listed as long as they are valid inputs for that parameter. For example, the **...** button for the **color** parameter to the Color module lists red, green, and blue.

Outputs Section

Name	Specifies the name of an output parameter. You cannot modify this field.
Type	Specifies the type of an output parameter. You cannot modify this field.
Destination	Displays the name of the tool (or tools) to which the output tabs are connected.
Cache	Specifies the number of results for this output of the tool that are eligible for caching. Cache can be “All Results”, in which all results are eligible for cache, “Last Result”, in which only the last value of the output is eligible, or “No Results”, in which no results from the module should be cached. (See “Cache Control: Executive” on page 215.)

Pushbuttons

The seven buttons at the bottom of the Configuration dialog box are labeled boxes that perform an action when you click on them. The buttons are:

OK	Applies new values and closes the box.
Apply	Saves newly entered values. Once you click on the **Apply** button, you can no longer restore previous values.
Expand	Causes all hidden inputs to be displayed in the Configuration dialog box. You can then unhide hidden tabs using the **Hide** toggle button.
Collapse	Causes all hidden input tabs to be removed from display in the Configuration dialog box.
Description	Displays a window with descriptions of the input and output parameters of the tool.
Restore	Restores previous values that were present when you opened the box or when you last clicked on the **Apply** button.
Cancel	Restores previous values that were present when you opened the box or when you last clicked on the **Apply** button, and then closes the box.

Revealing and Hiding Input Tabs

Most of the tools have more input tabs than are displayed when a tool icon is initially displayed. For example, the Isosurface module has six inputs, but when an Isosurface tool icon is initially placed on the canvas, only three input tabs are displayed. The displayed tabs represent those that are most frequently used.

To reveal the hidden input tabs:

1. Select the tool icon by clicking on it.
2. Select the **Reveal All Tabs** option from the **Edit** pull-down menu. (You can also use the **Ctrl+L** accelerator key.)

Once the input tabs have been revealed, they can be hidden.

To hide input tabs:

1. Select the tool icon by clicking on it.
2. Select the **Hide All Tabs** option from the **Edit** pull-down menu. (You can also use the **Ctrl+H** accelerator key.)

Hiding input tabs in this way will cause all tabs that are not connected to another module to be hidden.

If you wish to hide or reveal individual inputs, use the **hide** toggle button in the Configuration dialog box (see “Inputs Section” on page 108).

Using the Compute Module Configuration Dialog Box

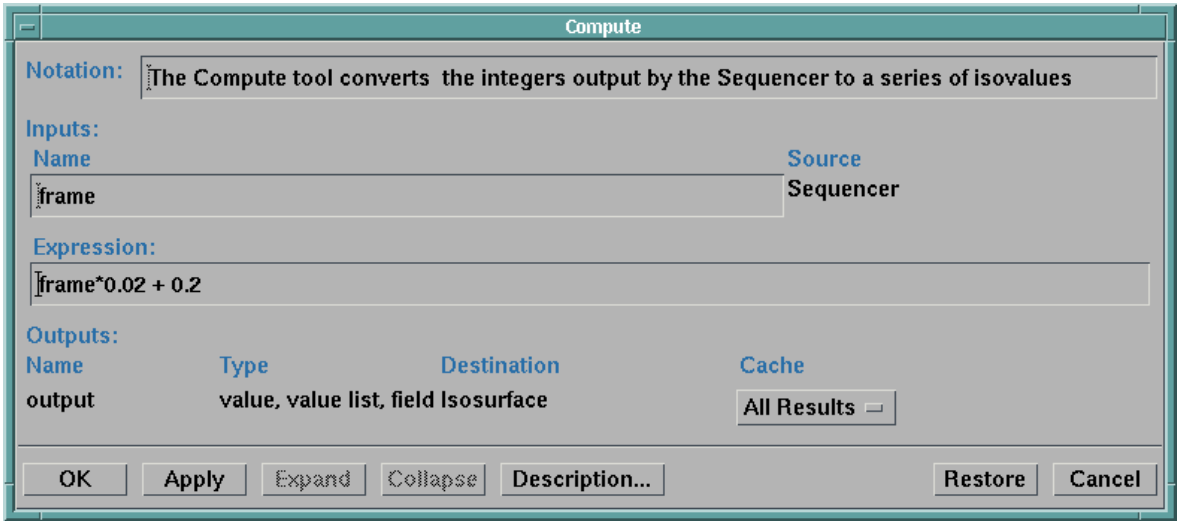
The Configuration dialog box for the Compute module, illustrated in Figure 51, differs in some respects from the other Configuration dialog boxes. It consists of:

Notation See “Notation Field” on page 108 for information.

Name These fields allow you to enter names for the parameters of the expression. By default, the names are labeled a and b, but you can change them to a name more relevant to the particular computation. (You can have fewer or more than two inputs to the Compute module; see “Adding and Removing Input and Output Tabs” on page 106.)

Source See information on Source fields on page 109.

Expression The expression is entered in this field. See Compute in *IBM Visualization Data Explorer User’s Reference* for more information.



Notation: The Compute tool converts the integers output by the Sequencer to a series of isovalues

Name	Source
frame	Sequencer

Expression: frame*0.02 + 0.2

Name	Type	Destination	Cache
output	value, value list, field	Isosurface	All Results

Figure 51. Typical Dialog Box for the Compute Module

Locating Tools: The Find Tool Dialog Box

The **Find Tool...** dialog box provides you with an easy way to locate tools on the VPE canvas. You may find this dialog box especially useful if you are editing a large visual program. The dialog box can also be used to locate transmitters and receivers by the names you give them. You open this dialog box by selecting the **Find Tool...** option from the **Edit** pull-down menu.

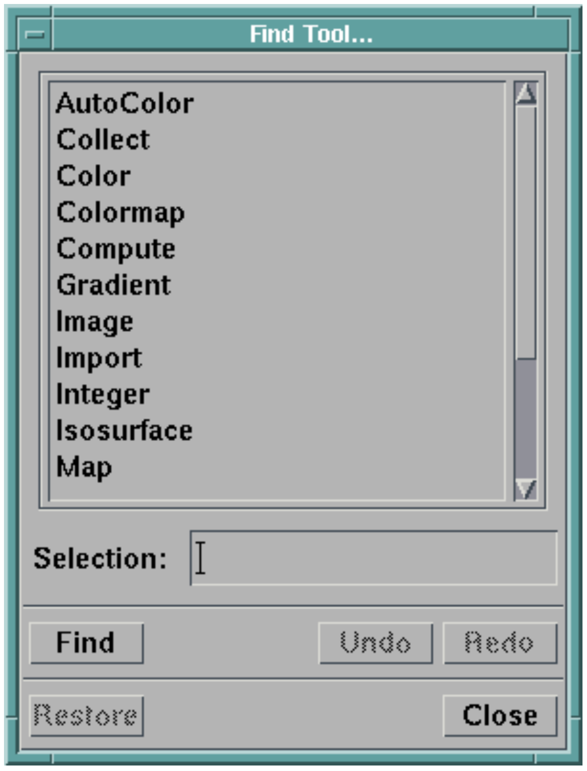


Figure 52. Find Tool Dialog Box

Figure 52 illustrates the layout of the Find Tool dialog box. The dialog consists of three parts:

Tool list Displays an alphabetized list of all the tools in the visual program currently displayed on the VPE canvas. If the visual program contains transmitters, receivers, probes or picks, then the tool name (e.g., Transmitter) is displayed instead of the user-supplied name.

Selection Displays the current tool to be located. When the dialog first appears, this field is blank. To change the selected tool either click on the desired tool name, or click on the **Undo** or **Redo** buttons, or type the name of the tool, transmitter or receiver directly in the selection text field. The next time you open the dialog box, the selection field will display the last selection you made.

Pushbuttons
Find initiates the search for the selected tool. **Undo** undoes the last find and updates Selection. **Redo** redoes an undone find and updates Selection. **Restore** restores the canvas to the location at the time the dialog box was opened. **Close** closes the dialog box.

To locate a particular tool on the canvas:

1. Click on the tool name.
2. Click on the **Find** button. This will initiate the search for the first occurrence of the tool icon on the canvas. When the tool is found, the portion of the canvas that is displayed may be updated to include the located tool icon.

Note: The located tool icon is selected.

If you wish to find another occurrence of the same tool, simply click on **Find** again. This can be repeated as many time as you desire. When no more occurrences of the tool can be found, a message is displayed. If you click on the **Find** button again, the search will be reset and the first occurrence of the tool icon will be located.

If the selected tool is a transmitter, receiver, probe, or pick, occurrences of the tool will be found independently of the user-supplied name. To locate a transmitter, receiver, probe, or pick tool by name:

1. Enter the name of the transmitter, receiver, pick or probe in the **Selection:** text field.
2. Click on the **Find** button.

You can initiate the search for a different tool at any point. If you wish to retrace your steps, the **Find** dialog provides an **Undo** button, allowing you to undo up to 10 previous searches. When you click on the **Undo** pushbutton, the name of the tool (or transmitter or receiver) that was previously located will appear in the **Selection:** text field and the canvas is updated to reflect the location of the tool icon.

The dialog box also provides a **Redo** button. This enables you to repeat a search that was undone with the **Undo** button.

If you wish to return the canvas to its former “state” (i.e., to the set of tool icons it displayed) prior to the first search, click on **Restore** button.

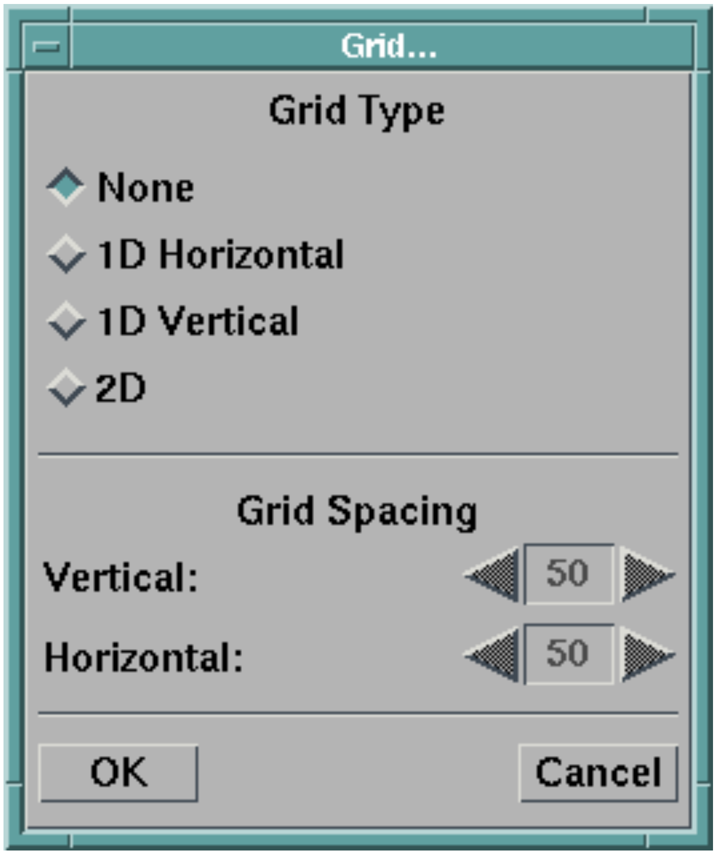
Note: Clicking on **Undo**, **Redo**, or **Restore** will deselect the tool that is selected in the Tools palette.

Customizing the VPE Window

Under the **Options** menu bar category are selections for customizing the window:

Tool Palettes Use as a toggle by clicking to close the palettes and make the working area on the canvas larger.

Grid... A dialog box, illustrated in Figure 53 on page 114, appears for you to enter new values.


Grid...

Grid Type

[✓ None] 1D Horizontal 1D Vertical 2D

Grid Spacing

Vertical: 50

Horizontal: 50

OK Cancel

Figure 53. Grid Dialog Box

The **Grid...** option allows you to specify whether the tools you place on the canvas automatically align on a grid pattern. You can enable the grid pattern by clicking on one of the following:

1D Horizontal
1D Vertical
2D

The default is none.

You can control the spacing of the grid pattern by changing the number of vertical and horizontal pixels. Specify how the tools are to be aligned on the grid by changing the alignment toggle buttons. For example, the dialog box in Figure 53 specifies grid spacing of 50 pixels, with the center of a tool being placed at a grid position.

Adding Comments to a Visual Program

For your own documentation purposes, you can add comments to your visual program. These comments are saved and restored as you save and restore the program.

To add a comment to the visual program:

1. Select the **Comment...** option from the **Edit** pull-down menu. A window opens with space for a large text field. If a comment has been entered previously, it is displayed in the text field.

2. Enter the desired comment in the text field. Edit the field the same way you edit any text field. This text field has multiple lines; you can generate line breaks using the Enter key, or type continuously and have the line breaks added automatically.

You can view these comments by using the **Comment...** option of the **Edit** pull-down menu in the VPE, or by using the **Application Comment** option of the **Help** pull-down menu in any primary window.

Adding Annotation to a Visual Program

You can also add annotation directly to the canvas. Select the **Add Annotation option** from the Edit pull-down menu. A cursor appears. Click on the canvas where you would like to place annotation. You can modify the text in the annotation by double-clicking on the annotation. A text-entry dialog appears.

By default, the annotation text is visible on the canvas. You can choose the **Hide Text** option on the text-entry dialog, in which case only a “marker” appears on the canvas.

Creating pages in the VPE

You can structure your visual program to make it more readable by using pages. A visual program can consist of a number of pages. Each page contains a set of modules completely disconnected from modules on other pages. Receivers and transmitters are used to connect modules on different pages.

To use pages, select the **Pages...** option in the Edit menu of the VPE. A cascade menu allows you to create an empty page, create a page containing the currently selected tools, delete the currently displayed page, or configure the page, that is, change its name or position.

Saving and Restoring a Visual Program

The **Save As...** and **Open...** options of the **File** pull-down menu use similar dialog boxes. A sample **Save As...** dialog box is illustrated in Figure 54 on page 116.

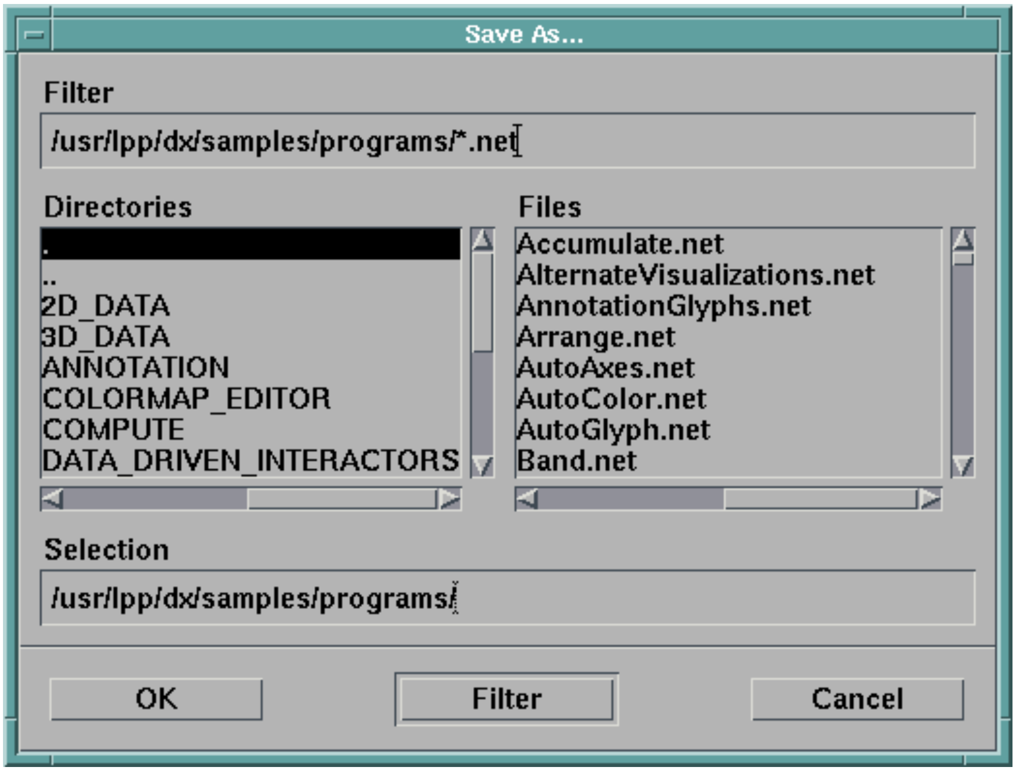


Figure 54. Save As Dialog Box

File Selection Dialog Boxes

The components of the dialog box are:

Filter Specifies the current search argument for files. You can broaden or narrow the scope of the files displayed in the **`Files`** area by changing the filter string. You can also use the filter to specify a directory in which to search for the files. For example, a filter of `/abc/*.net` displays all of the visual programs in the `abc` directory. Change the filter string by clicking on it and typing the new string. Because Data Explorer appends the `.net` extension to visual programs when it saves them, be sure to specify `.net` at the end of the filter string. To request smaller groups of files in the current directory:

1. Type standard file regular expression notation into the File Filter field. For example, type `ab*.net` to select all `.net` files whose names begin with `ab`.
2. Click on the **`Filter`** button at the bottom of the dialog box, or press Enter, to update the information shown.

To request files located in a different directory:

1. Use standard file regular expression notation to specify the directory to search. For example, to select all the `.net` files in the `/u/xyz` directory, you would change the filter string to `/u/xyz/*.net`.
2. Click on the **`Filter`** button at the bottom of the dialog box to update the information shown.

Directories Displays the directories in the current filter path. When you click on a directory, its name is displayed in the filter field; pressing the Enter key then applies that filter. You can traverse through the available paths by double-clicking on the paths displayed in this portion of the dialog box. The parent directory of the filter path can be reached by selecting and applying the directory name that ends with two periods (..) as a filter. As you change directories this way, the **Selection** box and **Files** section are updated accordingly.

Files Displays the files specified by the **File Filter** and the directory shown in the **Selection** area.

Clicking on a file name once will change the current selection. Double-clicking on a file name will select that file and proceed with the Open, Save As, or Load Macro operation.

Selection Displays the current file selection. When the dialog box first appears, **Selection** displays the current directory. To change the current selection, either click on the desired file in the **File** area or click on the **Selection** area and type the desired file name. The next time you open the dialog box, the **Selection** field displays the directory you most recently specified.

Pushbuttons **OK** approves the file name in the **Selection** area and proceeds with the Open, Save As, or Load Macro operation. **Filter** applies the filter string specified in the **File Filter** area. **Cancel** closes the dialog box. In the **Open...** and **Load Macro...** file selection dialog boxes, the **Comments** button lets you view any comments associated with the selected visual program file.

You can use the scroll bars provided on the right side and the bottom of the directories and file listings to view file and path names that are either too long or too numerous to fit in the available space.

Saving a Visual Program

When saving a visual program, Data Explorer saves the following files:

- The visual program, with a `.net` extension
- The configuration settings, with a `.cfg` extension

Data Explorer automatically appends `.net`, `.cfg` to the name you enter in the **Save As...** dialog box. However, if you enter your file name with a `.net` extension, Data Explorer does not add another `.net`. If you end the file name with any other extension, Data Explorer appends the `.net` extension to the extension you have specified. For example, a file named `abc.xyz` would be renamed `abc.xyz.net`.

An existing file can be saved in the following ways:

- Under the same name, replacing the previous version of the file
- Under a new name or directory, thus creating a new file and preserving the previous version.

Replacing a Previously Saved File: To save a program that has been named and saved previously, press **Ctrl+S** or select the **File** menu **Save** option. This replaces the previous version of the file. A named visual program has its name displayed in the title bar of the VPE window.

Saving a Visual Program as a New File: To save program as a new file:

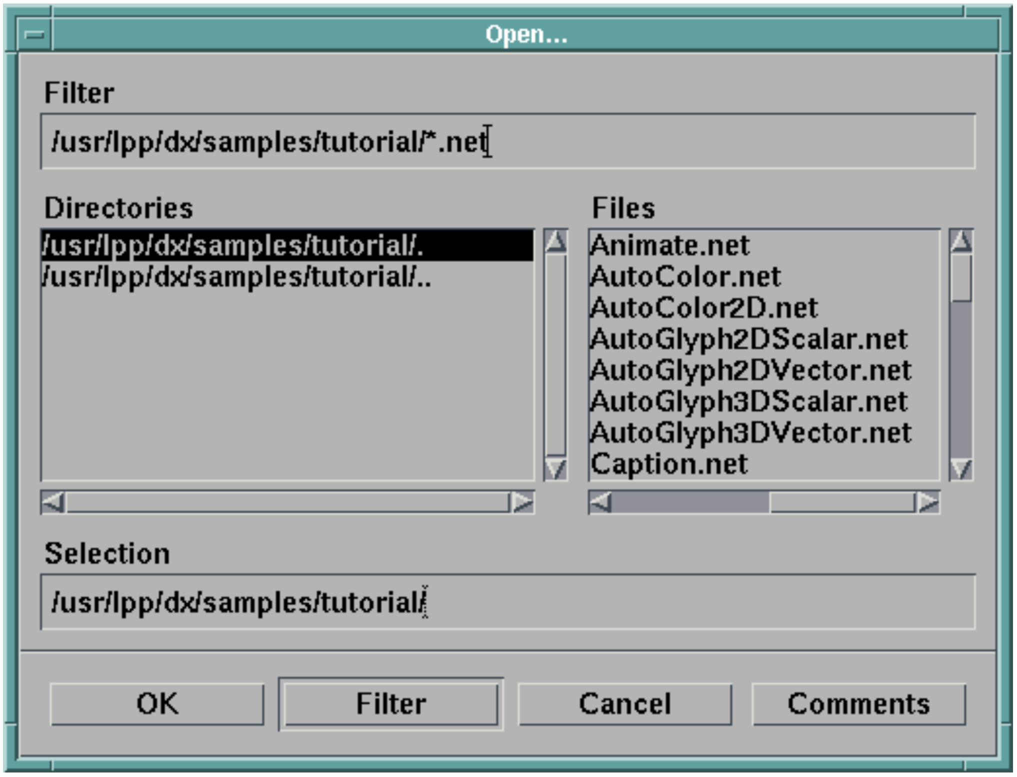
1. Select the **Save As...** option from the **File** pull-down menu. The **Save As...** dialog box appears (see Figure 54 on page 116). The **Selection** field displays the current file path.
2. Click on the **Selection** field and add the new file name. If you want to save the file to a different directory, change the file path in the field as well.
3. Press the Enter key or click on **OK**.

The file is saved and the **Save As...** dialog box disappears.

Restoring a Previously Created Program

To restore a previously created visual program, select the **File** menu **Open** option.

The **Open** dialog box appears (Figure 55).



/usr/lpp/dx/samples/tutorial/*.net

/usr/lpp/dx/samples/tutorial/.
/usr/lpp/dx/samples/tutorial/..

Animate.net
AutoColor.net
AutoColor2D.net
AutoGlyph2DScalar.net
AutoGlyph2DVector.net
AutoGlyph3DScalar.net
AutoGlyph3DVector.net
Caption.net

/usr/lpp/dx/samples/tutorial/

Figure 55. Open Dialog Box

It lists all file names in the current directory that are found through the specified file filter. A file can be selected in one of the following ways:

- Double-click on the file name under the **Files** heading.
- Click on the file name under the **Files** heading. The file name appears in the **Selection** text box. Click on **OK**.
- Click in the **Selection** text box and type the file name, then press Enter or click on **OK**.

Note that a complete file name must be specified in the **Selection** field to read in a file. A file name may be highlighted under the **Files** heading without appearing in the **Selection** field. To select the file, click on its name under the **Files** heading.

To see any comments that might be associated with a visual program before you load it, select a file by clicking on its name once or by entering its name in the **Selection** text box, then click on the **Comments** button. If the visual program has comments associated with it, they are displayed; otherwise, a message appears telling you that there are no comments.

To see a different list of files, change either the file filter or the file directory ("File Selection Dialog Boxes" on page 116).

When you open a file, its name is displayed in the title bar of the VPE.

6.3 Using the Colormap Editor

The Colormap Editor is a window that enables you to map colors to specified data values, the results of which are displayed in the visual image. In addition to color, the Colormap Editor also controls the mapping of *opacity* to data, which is the degree of the image's transparency in relation to its background. Maximum opacity shows the color calculated by the hue, saturation, and value fields; minimum opacity calculates colors so that the image is faintly visible in front of the background. In summary, the Colormap Editor enables you to:

- Control the range of data values over which the mapping occurs.
- Select the colors that are mapped to the range of values.
- Select the opacities that are mapped to the range of values.

When the **Colormap** stand-in from the **Special** category is connected to the **Color** tool as shown in the visual program fragment in Figure 56 on page 120, the combination can be used in place of the **AutoColor** tool.

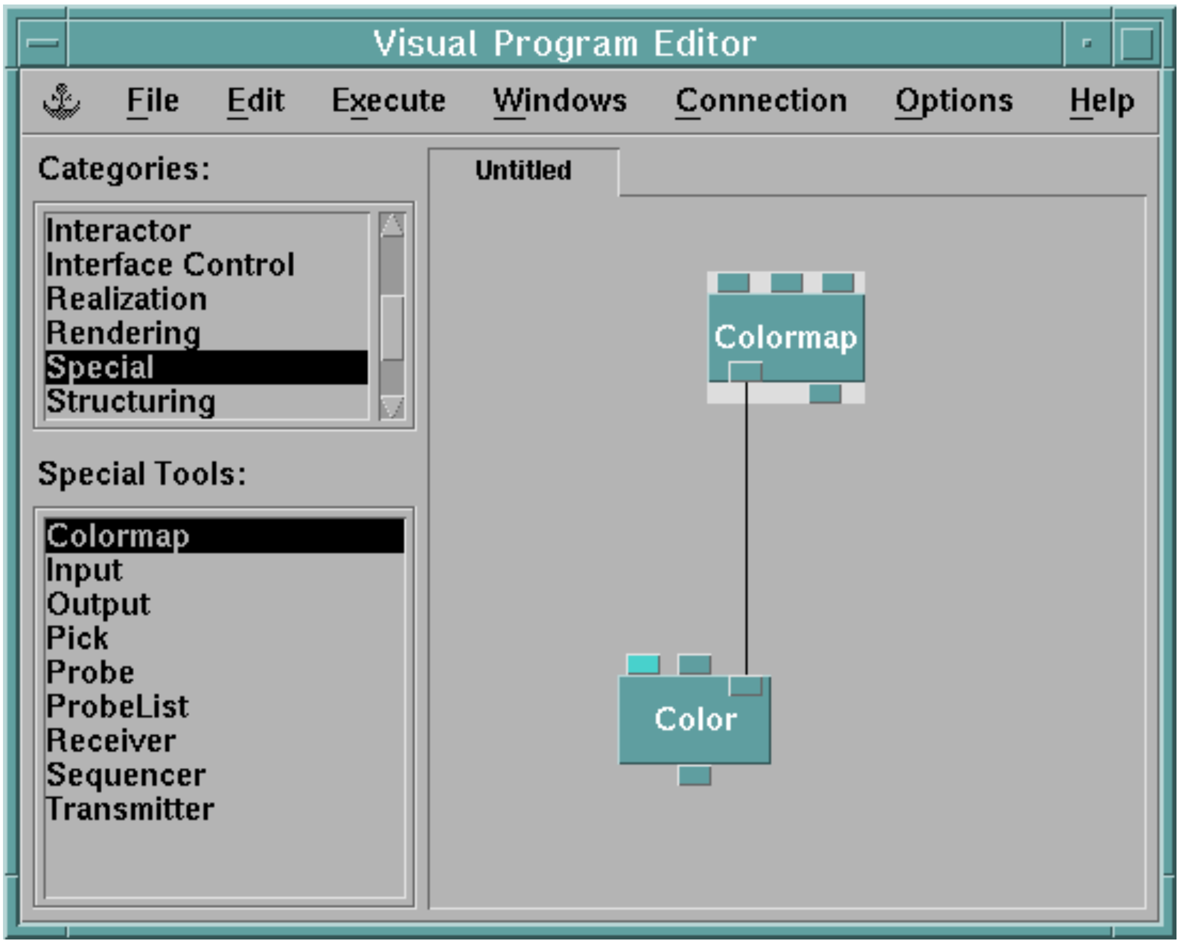


Figure 56. Fragment of Visual Program Using Colormap

To use the Colormap Editor:

1. Double-click on the Colormap tool in the VPE window or select either the **Open Selected Colormap Editors** option from the VPE or the **Open All Colormap Editors** from the Image window **Windows** menu.

   Note: From the VPE, this option is **Open Colormap Editor**. For this option to be available, the Colormap icon must be selected.

2. The Colormap Editor appears. Make necessary adjustments to values, as described in "Entering Values in a Colormap Editor" on page 121.

Figure 57 on page 121 illustrates the organization of the Colormap Editor window.

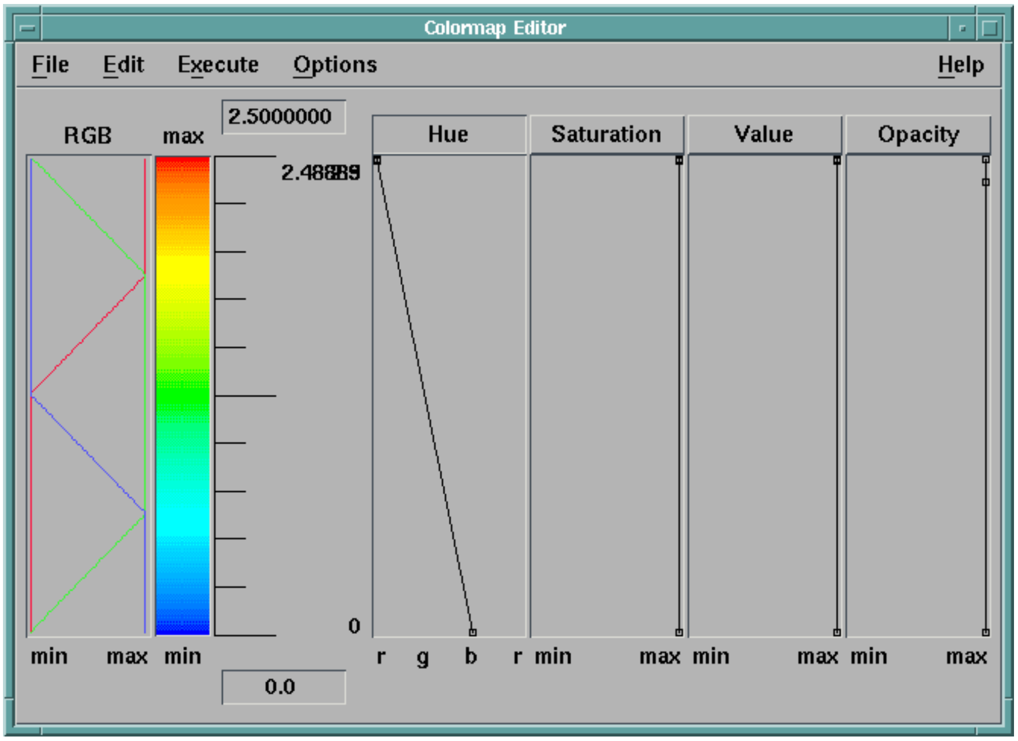


Figure 57. Colormap Editor

Entering Values in a Colormap Editor

The Colormap Editor specifies color in the hue, saturation, and value (HSV) color space. *Hue* refers to the color, for example, blue, red or yellow. The range of the hue goes from red to green to blue back to red again. *Saturation* refers to the purity of the color, and is a value between 0 and 1. A saturation of 1 is pure color; as saturation decreases, the color becomes more pastel, becoming white when saturation is 0. *Value* is the brightness of the color, and is a value between 0 to 1. A value of 1 is maximum brightness; as value decreases, the color becomes darker, becoming black when value is 0.

For a thorough understanding of color and the color elements of hue, saturation, value, and opacity, and other elements of computer graphics that might relate to the Colormap characteristics, you may want to refer to a computer graphics text.

You can display the Colormap Editor by selecting the **Open All Colormap Editors** option on the **Windows** menu of the Image window or by double-clicking on the Colormap tool in the VPE window. The Colormap Editor displays default settings for each of the three HSV color space parameters, as shown in the first three boxed areas on the right hand side of the window. These three areas, labeled **Hue**, **Saturation**, and **Value**, each work independently of one another. As you change their values, the RGB boxed area at the left of the window changes automatically to correspond.

The **Opacity** area, located on the far right hand side of the Colormap Editor window, works in a similar way. As you make changes in the opacity area, the background bar (located to the right of the RGB bar) reflects your work. It shows your adjustments to the opacity of the image in relation to the background colors. By default, the background bar appears as two vertical stripes. However, if it is easier for you to judge the colors of the image and background with a checkerboard-style bar, select the **Set Background Style to Checkboard** option on the **Options** menu (see “Colormap Options Menu” on page 170).

In order to perform certain operations on an area, it must be selected. To select an area either click on the area’s label or click in the area itself. Only one area can be selected at a time. When an area is selected, its label is depressed.

The range of data values onto which HSV and opacity values are mapped is controlled by the **min** and **max** fields located near the bottom and top of the Colormap Editor window. By default, **min** is set to 0, and **max** is set to 100. You can change this range to values more appropriate for your data by clicking on either field, typing the new value, and pressing the Enter key.

Control points are used to define the value of hue, saturation, value, and opacity for a given data value. The number and position of control points can be different in each of the areas. The control points appear as small squares on the vertical scale marks in each of the four areas.

Adding Control Points

Control points can be added to an area using one of four different methods:

- double-clicking directly in the area,
- using the **Add Control Points...** dialog box,
- using the **Generate Waveforms** dialog box, or
- copying and pasting control points from another area.

To add a new control point by double-clicking, place the cursor on the location where you want the new control point, then double-click. The values between control points are linearly interpolated by the Colormap. If a new point is added as the bottom- or top-most point on the line, the new line continues vertically from the new point to the **min** or **max** value, respectively. When a new control point is added, its data value is displayed by default.

To specify exact values for new control points, click on the **Add Control Points...** option on the **Edit** menu. The **Add Control Points** dialog box appears, as illustrated in Figure 58 on page 123. The **Add Control Points...** dialog allows you to specify values using two steppers. The “Data value” stepper allows you to specify a control point value between “min” and “max”. The second stepper displayed in the dialog will reflect that area (Hue, Saturation, Value, or Opacity) is currently selected in the Colormap Editor. For example, if Saturation is the selected area, the dialog will display steppers for “Data value” and “Saturation value”. The value for Hue, Saturation, Value, or Opacity can be a value between 0 and 1. Use the **Add** button to add the control points to the selected area in the Colormap Editor.

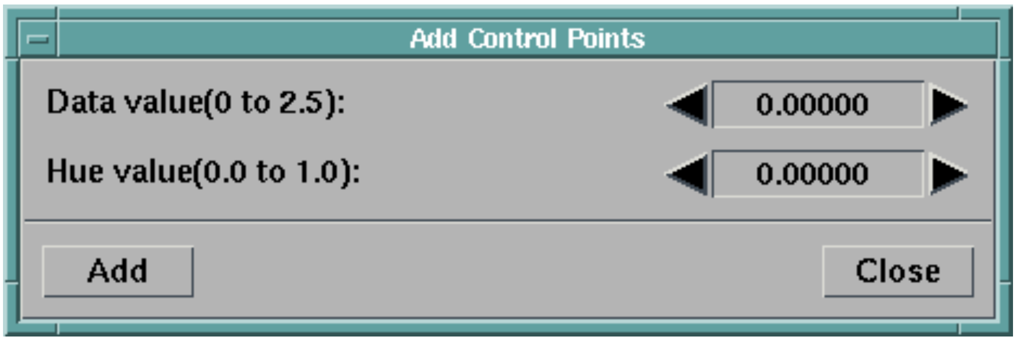


Figure 58. Colormap's Add Control Points Dialog Box

Selecting Control Points

Control points can be selected by doing one of the following:

- Select a single control point by simply clicking on it once.
- Select a group of control points by clicking on a point in the selected area and dragging the cursor around the desired points.
- Select all of the control points in an area by using the **Select All Control Points** option under the **Edit** menu.

Note: A control point is selected automatically when it is created. When one control point is created, all other previously selected points in that area are automatically deselected.

Deleting Selected Control Points: To delete selected control points, you can do one of the following:

- Double-click on each of them, one at a time.
- Choose the **Edit** menu, then click on the **Delete Selected Control Points** option.

Moving Control Points

To move a control point, simply drag it to the desired location. Control points cannot be moved past each other; this facilitates the creation of step functions. They can be moved as a group by doing the following:

1. Draw a selection box around the points you want in the group.
2. Position the mouse pointer on any one of them and drag it to the desired location.

All of the control points move together within the constraints of the unselected points above and below.

The movement of control points can be constrained either horizontally or vertically by selecting the **Constrain Horizontal** or **Constrain Vertical** option from the Colormap **Edit** menu (see “Colormap Edit Menu” on page 169). By constraining horizontally after adding a precise control point, you can move the point to change the color or opacity mapped to specific value, without changing the value itself.

Creating Waveforms

To create waveforms, select the **Generate Waveforms** option from the **Edit** pull-down menu. The **Generate Waveforms** dialog box appears (Figure 59 on page 125).

This dialog box allows you to:

- Choose the shape of the waveform from an options box. Waveforms can be step, square, or sawtooth.
- Choose the range of the waveform from an options box. “Full” creates a waveform that runs the full length of the selected area. “Selected” creates a wave that runs the distance between two selected control points in the specified area.
- Specify the number of steps to be created in the range of the waveform by using the stepper. The number of steps specified can be between 2 and 100.

Copying and Pasting Control Points

Control points can be copied and pasted from one Colormap area to another, using the **Copy** and **Paste** options of the **Edit** window.

1. Select the control point or control points you wish to copy.
2. Click on **Copy**.
3. Select the area to which you wish to copy the control points and then click on **Paste**.

Display Control Point Values

The data values of control points are displayed by default. You can control which data values are displayed, using the **Display Control Point Data Value** cascade menu in the **Options** menu. If “off” is specified, no data values are shown. If “selected” is chosen, only the data values for the selected control points are shown, and if “all” is selected, the data values for all control points in the selected area are shown.

Axis Display

You can control how the Colormap Editor axis is displayed by using the **Options** menu **Axis Display...** option. You have three choices for the display: **Ticks** (the default), **Histogram**, and **Log Histogram**. **Histogram** will cause the histogram of the data to be displayed. **Log Histogram** will cause the log of the histogram to be displayed. If the Colormap Editor is not data-driven, these two options will be grayed-out. The number of histogram bins can be controlled using the **Edit** menu **Number of histogram bins...** option.

Changing the name of the Colormap Editor

Every Colormap Editor is given the default name of “Colormap Editor” in the box across the top of the window. If you want to customize the name of the Colormap Editor, you can do so by clicking on the **Options** menu **Change Colormap Name...** option and entering a new name in the dialog box that appears.

Note: You can also change the name of the Colormap Editor by using the **title** parameter in the Colormap tool.

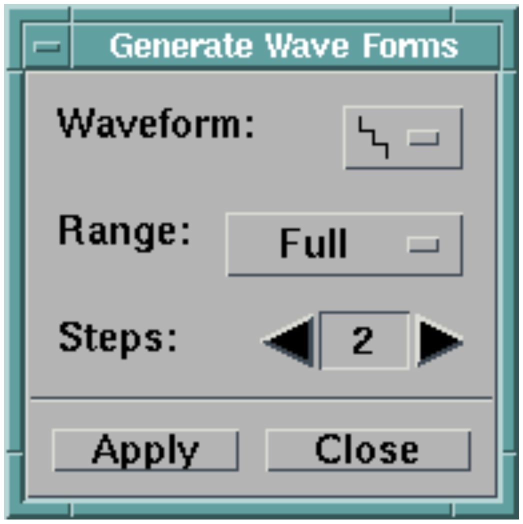


Figure 59. Generate Waveforms Dialog Box

Saving and Loading Color Maps

You can save a color map by using the **Save As...** command from the **File** pull-down menu). You can then make the “new” color map part of any visual program. To access it, use the **Open** command from the **File** pull-down menu of the Colormap Editor menu bar. Note that saved color maps may also be imported (see Import in *IBM Visualization Data Explorer User’s Reference*) and passed directly to the Color module.

Using Data-Driven Colormap Editors

The Colormap Editor may be *data-driven*, meaning that its attributes (e.g., minimum and maximum) can be set by connecting the output of a tool to the input of the Colormap tool in the VPE or by typing a value into the Colormap configuration dialog box, instead of into the Colormap Editor itself.

If the Colormap Editor is data-driven, the information transmitted via the connections or set in the Configuration dialog box overrides values set in the Colormap Editor.

Data-driven Colormap Editors allow you to create color maps that are appropriate for a variety of input data sets without the need to reset the minimum and maximum of the color map.

The Colormap tool has a data input to which an input data field may be connected. In this case, the Colormap Editor is automatically set so that the minimum is the minimum of the data set and the maximum is the maximum of the data set. However, if you would like to have more control over the exact values that are used, the Colormap tool allows you to specify the minimum and maximum directly through other input tabs that are by default hidden. You can also pass a color map or opacity map directly to the Colormap tool. The inputs for the Colormap tool are summarized in the corresponding module description in *IBM Visualization Data Explorer User’s Reference.*

Each time an input to a data-driven Colormap Editor is changed (e.g., by importing a new data set), the interactor is reexecuted, updating its attributes.

Chapter 7. Graphical User Interface: Control Panels, Interactors, and Macros

7.1 Using Control Panels and Interactors

As you create a visual program, you may have inputs whose values are subject to frequent change. You can use interactors as an easy method for controlling those input values. *Interactors*, which appear only in Control Panels, are the interactive devices that you use to manipulate inputs to a visual program in order to change the image that is produced (see “Using Interactors” on page 142 for detailed descriptions).

Interactor stand-ins are used to indicate which input to a module a given interactor is to control. While you are building the network in the VPE, you select interactor stand-ins from the tool palettes and place them on the canvas, as you do with other tools. Like any tool, the output of an interactor stand-in can be connected to more than one input. Interactor stand-ins are named, in general, after the type of data they output:

Integer stand-ins
Represent interactors that output whole numbers.

Scalar stand-ins
Represent interactors that output real numbers.

String stand-ins
Represent interactors that output text strings.

Value stand-ins
Represent interactors that output scalars, vectors, and tensors.

Vector stand-ins
Represent interactors that output vectors.

Integer list stand-ins
Represent interactors that output integer lists.

Scalar list stand-ins
Represent interactors that output scalar lists.

String list stand-ins
Represent interactors that output string lists.

Value list stand-ins
Represent interactors that output value lists (e.g., vector and scalar lists).

Vector list stand-ins
Represent interactors that output vector lists.

Selector stand-ins
Represent interactors that output values and strings, representing a choice of one from many.

Selector list stand-ins
Represent interactors that output values and strings, representing a choice of none, one, or more among many.

FileSelector stand-ins
Represent interactors that output both a fully qualified path name and an individual file name.

Reset stand-ins
Represent interactors that output one value when executed the first time after being set and another value thereafter.

Toggle stand-ins
Represent interactors that output one of two values.

Data Explorer allows the visual programmer to associate comments with each Control Panel. To access these comments, use the **On Control Panel** option of the **Help** pull-down menu in the Control Panel about which you want to learn. If no comments exist for the Control Panel, the **On Control Panel** option is grayed out.

Building Control Panels

Figure 60 illustrates the organization of a Control Panel.

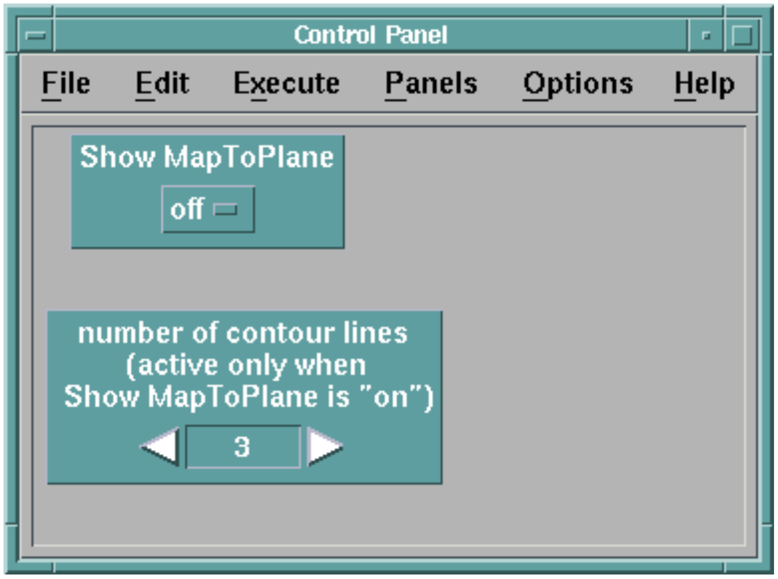


Figure 60. Control Panel Window

The menu bar, discussed in “Control Panel Menu Bar” on page 162, contains categories of available menu options. The open area is called the *layout area*.

You can create any number of Control Panels for one visual program, and you can also place a single interactor in multiple Control Panels. The configuration of a Control Panel and the values of the interactors are saved when you save the visual program. You can also customize Control Panels, and save or restore them independently of the visual program.

If you are going to control a tool input through an interactor in a Control Panel, then you must first connect an interactor stand-in to that input in the VPE. Using a stand-in as a tool input is an alternative to using the module’s Configuration dialog box every time you want to change the value of the input.

Placing Interactors in a New Control Panel

To place interactors in a new Control Panel:

1. On the VPE canvas, select the interactor stand-ins you want in the Control Panel (other tools can be selected as well).
2. Click on the **New Control Panel** option of the **Windows** menu. This causes a new Control Panel to appear with the selected interactors in the layout area. Each interactor is labeled with the name of the tool to which its output is connected, unless its output is connected to more than one tool or not connected to any tool, in which case the interactor is labeled with the interactor type (e.g., an integer interactor is labeled with “Integer”). You can also double-click on one of the selected interactor stand-ins to create a new Control Panel automatically.

When a new Control Panel is created with the selected interactors, the interactors are placed in a vertical column in the order in which their stand-ins were placed on the VPE canvas.

Note: If you select a group that includes tools other than interactor stand-ins, only the interactors appear in the Control Panel. Therefore, the quickest way to place all the visual program’s interactors in one Control Panel is to use the **Select All** option of the **Edit** pull-down, then select the **New Control Panel** option of the **Windows** pull-down menu.

Adding Interactors to an Existing Control Panel

To add an interactor to an existing Control Panel:

1. Open the Control Panel.
2. Select the interactor stand-in on the VPE canvas.
3. In the Control Panel window, click on the **Add Selected Interactor(s)** option of the **Edit** menu.
4. Move the cursor to where you want to position the tool in the Control Panel. Note that the style of the cursor changes when you move it onto the panel. This is similar to how tool icons are placed on the VPE canvas. Exact placement of the interactor depends on the grid settings (see “Changing the Alignment of Interactors in the Control Panel” on page 134).
5. Click the mouse. The interactor appears at the specified location.

Alternatively, you can use “drag and drop” to add an interactor to an existing Control Panel:

1. Select the interactor stand-in on the VPE canvas.
2. Press the middle mouse button while the cursor is positioned on the stand-in icon.
3. Drag the cursor to the Control Panel and release the mouse button.
4. The interactor appears at the new location.

You can add more than one interactor at a time to a Control Panel. To do this, select multiple interactor stand-ins on the VPE canvas, then select **Add Selected Interactor(s)** in the desired Control Panel. After doing this, you can use the mouse to place the interactors in the Control Panel one at a time. They are placed in the Control Panel in the same order that they were placed initially onto the VPE canvas. Similarly, you can drag and drop multiple stand-ins from the visual program to a Control Panel. You can also drag and drop interactors from one Control Panel to another, as long as both are associated with the same Data

Explorer session. For more information on drag and drop, see “Moving and Copying Tools” on page 105.

You can put the same interactor in more than one Control Panel and in the same Control Panel more than once. For example, you may want to have one Control Panel that contains *all* the interactors for a visual program, and another that contains only the most frequently used interactors. You can also place multiple instances of an interactor, with different styles or step size increments, in one Control Panel. This provides both coarse and fine control over a parameter value. The user interface ensures that each instance of the interactor is consistent. If you change a value in one instance, it changes in the others.

Selecting, Moving, and Deleting Interactors

To select an interactor, click on it.

To select a group of interactors, use one of the following methods:

- Hold down the Shift key and click on each interactor in turn.
- Position the mouse pointer on the canvas near a corner of the group and drag the mouse to draw a selection box around the interactor you want. To select an interactor, you must completely enclose it in the selection box. As an interactor is encompassed by the selection box, it is highlighted (indicating that it is selected).

To deselect an interactor, shift-click on it.

To deselect a group of interactors, use one of these methods:

- Shift-click on each interactor.
- Shift-drag to draw a box around the interactors you want to deselect. As a selected interactor is encompassed in the box, it becomes unhighlighted. Release the button to deselect the interactors.

Note: Clicking on an empty part of the layout area deselects all selected interactors.

To move an interactor:

1. Depress the mouse button on the interactor and drag it to the desired location. While the mouse button is depressed, an outline of the interactor follows the motion of the mouse, but the interactor remains in the original location.
2. Release the mouse button. The interactor moves to that location.

To move a group of interactors:

1. Select a group of interactors to be moved.
2. Position the mouse pointer on any member of the group and drag it. An outline of each interactor appears and follows the mouse.
3. Release the mouse button. The interactors move to the new location.

To delete one or more interactors:

1. Select one or more interactors to delete.
2. Press the **Ctrl+Delete** accelerator key or click on the **Delete** option of the **Edit** menu.

If you delete an interactor stand-in from the VPE, the interactor in the Control Panel is also deleted. However, deleting the interactor in the Control Panel does not affect what is displayed in the VPE.

Changing the Size of an Interactor

You can change the size of interactors in a Control Panel. Some interactors (e.g., Selector) resize automatically, depending on their contents, but others (e.g., String) do not. You can resize any interactor by pressing the control key (Ctrl) and the left mouse button.

Locating Interactor Stand-ins

As you are building and modifying visual programs and Control Panels, you may find it desirable to locate that interactor in a Control Panel that corresponds to a specific interactor stand-in. To locate an interactor corresponding to a stand-in:

1. Select the desired stand-in in the VPE by clicking on it.
2. Select the **Edit** menu **Show Selected Interactor(s)** option in the Control Panel.

The interactor corresponding to the stand-in will become selected. If no interactors in the Control Panel are associated with the selected stand-in, the **Show Selected Interactor(s)** option will be grayed-out. If you have more than one Control Panel and you are unsure which of them contains the interactor corresponding to the stand-in, Step 2 (above) can be applied to each Control Panel. Alternatively, you can double-click on the stand-in. This will highlight the corresponding interactor.

To locate a stand-in corresponding to an interactor:

1. Select the interactor in a Control Panel by clicking on it.
2. Select the **Edit** menu **Show Selected Tool** option in the Control Panel.

The stand-in corresponding to the interactor will be selected. If the stand-in is not in the currently displayed portion of the visual program, the display will be updated so the selected stand-in is visible.

Deleting Control Panels

To delete a Control Panel:

1. Delete the interactors in the Control Panel.
2. Click on the **Close** option of the **File** menu.

Saving and Restoring Control Panels

The **Program Settings...** option of the anchor window allows you to save your own configuration of the Control Panel(s) independently of the rest of the network. You can save the values of the Control Panels as well as the configuration of all of the stored Control Panels for the current visual program.

Select the **Program Settings...Save As** option on the **File** menu anchor window bar. You can retrieve Control Panels in the anchor window by using the **Program Settings...Load** option.

If you have made changes that you do not want to keep, click on the **Program Settings...Load** option of the **File** menu, and select the file again without saving. This procedure restores the original configuration.

Customizing a Control Panel

This section describes the customization that can be done while building the Control Panel from the VPE window or while actually viewing the image in the Image window.

The Control Panel **Options** menu provides several possibilities for further customizing the Panel and its interactors, as described in "Control Panel Options Menu" on page 164.

Changing the Name of a Control Panel

Every Control Panel is given the default name of "Control Panel," as shown in the title box across the top of the window. If you want to customize the name in any particular Control Panel, you can do so by clicking on the **Change Control Panel Name...** option on the **Options** menu and entering a new name in the dialog box that appears. The new name can contain any number of characters including any letter, number, symbol, or space that you find on the keyboard.

If you have several Control Panels in your visual program, you should assign names to them. Data Explorer allows you to open each one individually, by name, from a Control Panel, Image window, and VPE. To open a Control Panel by name from any of these three primary windows, do the following:

1. Select the **Open Control Panel by Name** option from the **Windows** pull-down menu in the VPE and Image window, or from the **Panels** pull-down menu in the Control Panel. This reveals a cascade menu with a list of the existing Control Panels.
2. Click on the name of the Control Panel you wish to open. The desired Control Panel appears.

Adding Comments to a Control Panel

If other people are going to use the visual programs you create, it may be desirable to document how the interactors are used. You can associate comments with the Control Panel to describe how it uses the interactors to control input values in the visual program.

To add comments to a Control Panel:

1. Select the **Comment...** option from the **Edit** pull-down menu in the Control Panel. A dialog box appears, with a large text field in which you can type the comments. If a comment has been entered previously for this Control Panel, it is displayed in the text field.
2. Enter your comments in the text field, editing the same way as with any text field. This text field has multiple lines; you can break the lines using the Enter key, or allow them to flow automatically as you type.
3. Click on **OK** to store the comments.

These comments can be viewed, but not edited, by using the **On Control Panel** option of the **Help** pull-down menu in the Control Panel. To edit the comments, you must use the **Comment...** option of the **Options** pull-down menu.

Changing the Alignment of Interactors in the Control Panel

You can specify whether the interactors you place in the Control Panel automatically align on a grid pattern. To do this, select **Grid...** from the **Options** pull-down menu. The **Grid** dialog box appears; it works the same way as it does in the VPE (see “Customizing the VPE Window” on page 113).

Changing the Interactor Style

A particular default interactor might not be the most desirable style for your particular application. For some interactor types, you can change this in a Control Panel at any time by using the following procedure:

1. Select the interactor.
2. Click on the **Edit** menu **Set Style...** option.

   Note: Be sure to highlight the interactor in the panel before selecting this option. Otherwise, most of the options will appear grayed-out. If the interactor is not a type whose style can be changed, the style option will remain grayed-out. A cascade menu appears for you to choose a new style.

Resizable Interactors.

You can change the size of the interactors in a Control Panel. Some interactors (e.g., Selector) resize automatically; others (e.g., String) do not. To resize any interactor, press the Control key and then drag the border of the interactor.

Changing the Interactor Dimensionality

When vector and vector list interactors are created, by default, their dimensionality to set to 3. Dimensionality can be changed using the **Edit** menu **Set Dimensionality** option.

Changing the Interactor Layout

When an interactor is created, its layout is vertical, that is, the interactor label is placed at the top of the interactor. Data Explorer allows you to choose between this vertical layout and a horizontal layout. The horizontal layout places the label on the left side of interactor.

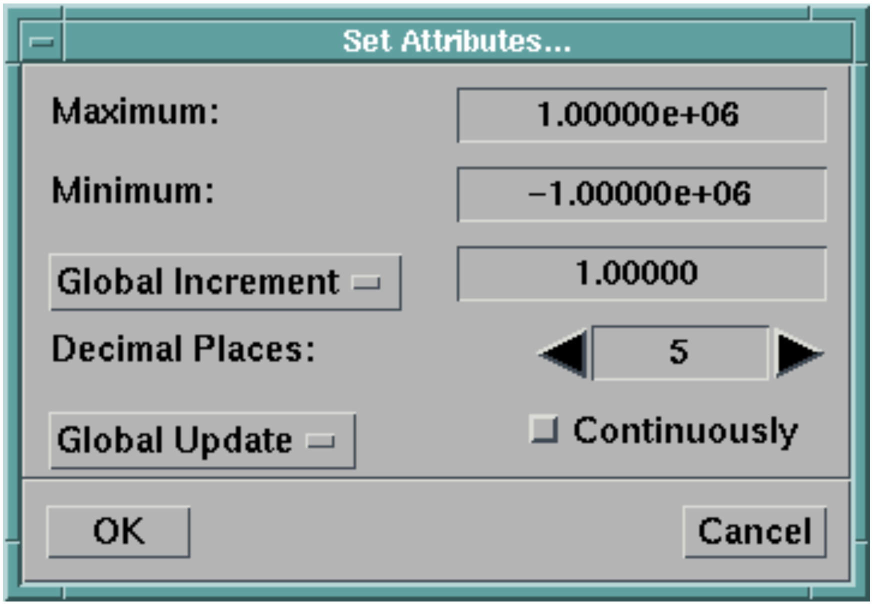
To change the layout of the interactor:

1. Select the interactor.
2. Click on the **Edit** menu **Set Layout...** option. A cascade menu appears for you to choose the layout you desire.

Setting Interactor Attributes

You can have several instances of the same interactor. Each instance can be a different style (stepper, dial, or slider) and can have a different increment value. However, all instances of an interactor have the same value and the same minimum and maximum limits. As you change the value or range of one instance, the other instances of that interactor are automatically updated.

To change the *range* of values of an interactor, select the interactor, and select the **Set Attributes...** option in the **Edit** menu; or double-click on the interactor in the Control Panel. The **Set Attributes...** dialog box (Figure 61 on page 135) appears.


Set Attributes...

Maximum: 1.00000e+06

Minimum: −1.00000e+06

Global Increment: 1.00000

Decimal Places: 5

Global Update

☐ Continuously

Figure 61. Set Attributes Dialog Box

In this box you can:

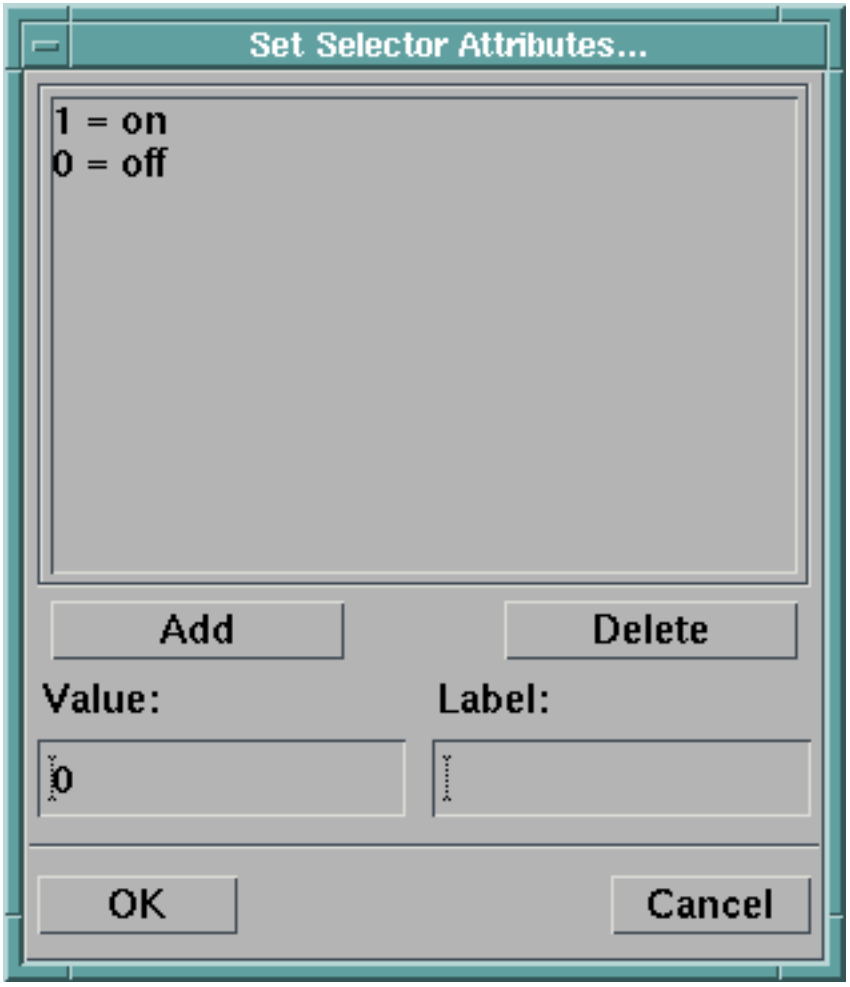
- Set maximum and minimum values.
- Change the step increment.
- Change the number of decimal places displayed. (In the case of an integer interactor, the decimal place field is disabled.)
- Choose whether to update the image continuously as you change the interactor values, or to update only when the mouse button is released. This applies only when `Execute on Change` is enabled.

  Note: Many interactor types can also be data-driven, meaning that their attributes are derived at run time from data in the visual program. See "Using Data-Driven Interactors" on page 147.

The increment and update options can be applied to the either just the current instance or to all instances of the interactor. To affect all instances, click on the option box for the attribute you want to set (increment or update), and select the Global option from option menu. To affect only the current instance, select the Local option. Having multiple instances of an interactor with different increments allows coarse and fine controls. For example, you may want one instance of a scalar interactor to change by increments of 1.0, another instance by 0.10, and a third instance by 0.01. Note that the "global" option will not override interactors which have been explicitly set to "local".

Vector interactors The `Set Attributes...` dialog box for vector interactors has an additional field, `Selected Component`, included at the top of the box. You can use this field to set different attributes for the different components of a vector. Do this by changing the component number with the stepper, and setting the attributes desired for that component. Repeat this process for all components of the vector.***:***
To assign common attributes to all components of a vector interactor, set the option box at the top of the dialog box to `All Components`. When you do this, the component stepper is disabled, and any attributes you set are applied to all components of the vector.

Setting Selector and SelectorList Interactor Attributes The Set Attributes dialog box for the **Selector** and **SelectorList** interactors (Figure 62 on page 136) differ from the dialog box for other types of interactors because the behavior of those interactors is different from other types of interactors.***:*** The Selector and SelectorList interactors are similar to an option menu, with the current choice(s) displayed by the interactor. (The Selector offers a one-of-many choice; the SelectorList, a choice of none, one, or more among many.) Each choice on the interactor represents a pair of outputs: a value and a string. The string is what appears as the choice on the interactor. The value can be a string, integer, scalar, vector or matrix. All the values must be of the same type. By default, the values are integers. A value is associated with a string using the **Set Attributes...** dialog box (Figure 62).



1 = on
0 = off

Value: 0

Label:

Figure 62. Set Attributes Dialog Box for a Selector Interactor

You can use the Selector or SelectorList interactor for many purposes. A common use is as a switch control in your visual program. You can use an integer output, for example, as input to the Switch module to switch easily among several objects. You can use a string output, for example, as input to the Select or Import modules, allowing you to easily select different members or data file names. You can also use a string output as input to the Caption module to annotate the image with the current selector setting. A discussion on how to use the Selector and SelectorList interactors can be found in “Selector and SelectorList Interactors” on page 146.

The default choices for the Selector and SelectorList interactors are:

- 1, “on”
- 0, “off”

To modify the choices, do the following:

1. Open a Control Panel with the selector interactor in it.
2. Open the selector’s **Set Attributes...** dialog box by double-clicking on the interactor, or by selecting the interactor and then choosing the **Set Attributes...** option in the **Edit** pull-down menu. A dialog box appears, similar to the one shown Figure 62 on page 136.
3. The procedure for modifying the choices in this list is similar to that for modifying the list elements in a list interactor (see “List Interactors” on page 145). To enter a value in this interactor click on the **Value** box, enter a value, and press the Enter key. Then click on the **Label** box, type a string, and press the Enter key. For more information about how to modify, add, and delete elements, see “List Interactors” on page 145.

   Note: If you wish to change the type of the values in the value field, for example, entering values other than integers when initially configuring the interactor, you must first delete all the entries in the **Set Attributes...** dialog before entering new ones. This is required since the type of all values must be the same.

4. When you are finished modifying the choices, click on **OK** to apply the changes.

Because the selector interactor yields both the value and the string as outputs, you can use either output, or both, in a visual program. In the selector stand-in, the output on the left is the value, while the output on the right represents the string.

Changing the Label on the Interactor: The default label on any unconnected interactor is the name of the interactor stand-in. If the interactor is connected to one input, the default name of the interactor is the name of the tool followed by the input parameter name. If the interactor is connected to more than one input, its default name is the name of the interactor stand-in. You can change the default name by doing the following:

1. Click on the **Set Interactor Label...** option on the Control Panel **Edit** menu.
2. Enter a new name in the dialog box that appears (Figure 63).

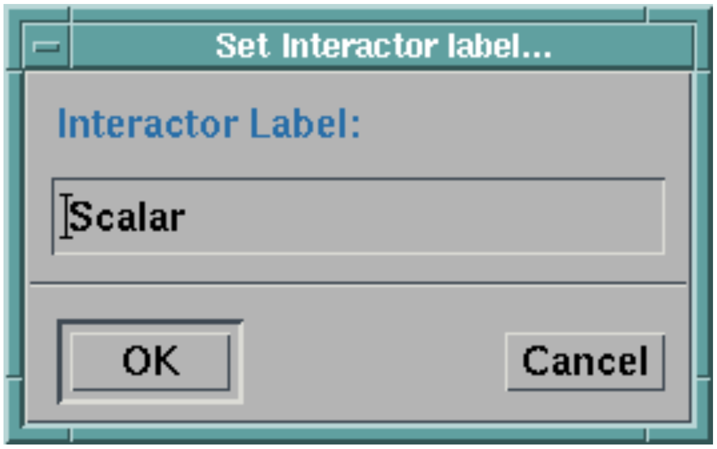


Figure 63. Set Interactor Label Dialog Box

The new name can contain any number of characters, including any letter, number, symbol, or space that you find on the keyboard. (If you want a blank label, enter "\0" for the name.)

The interactor label can have multiple lines: type "\n" where you want a line to break. For example,

```
First Line\nSecond Line
```

Setting Toggle and Reset Attributes: You can set the output of Toggle and Reset interactors for both their "button down" (set) and "button up" (unset) states.. The output value can be string, integer, scalar, or vector, and the set and unset outputs do not have to be of the same type.

Control Panels as Dialog Boxes

It is possible to customize Control Panels so that they appear as dialog boxes. This is intended for building applications to be used in `-image` or `-menubar` mode (i.e., with the Image window or menu bar as the anchor window). The appearance of the dialog box can be modified with the options **Label** and **Separator** under **Add Element** in the **Edit** pull-down menu of the Control Panel. These will add the specified element to the panel at the point where the mouse cursor is positioned. Separators can be made vertical using the **Set Layout** option of the **Edit** menu. The size of the separator can be controlled using the mechanism described in "Resizable Interactors." on page 134. The color and font of labels can be specified using the dialog box that appears when the label is created. These can be changed by selecting the label and choosing **Set Attributes** from the **Edit** menu of the Control Panel.

Once the Control Panel has been created, select **Dialog Style** in the **Options** pull-down menu to create the dialog box. The size of the box will vary to accommodate the interactors. The empty canvas to the right of the interactors will be truncated. The placement of the interactors will also change if the box size is changed, to maintain the same relative positions. In `-editor` mode, the **Close** button returns to an editable Control Panel. In `-image` or `-menubar` mode the, **Close** button closes the dialog box. To enable a "dialog style" dialog box, save the visual program with the Control Panel in dialog-style format.

Control Panel Access, Groups, and Hierarchies

You may wish to organize your Control Panels into groups or hierarchies depending on how the interactors in the Control Panels relate. For example, you may have a few Control Panels that are tightly related and wish to have them treated as a group. Data Explorer provides you with the means of placing these Control Panels into a group so that they can be opened together.

If you have a master Control Panel that should be open before any other controls, Data Explorer provides you with the capability of restricting access to these Control Panels from any Data Explorer window except the master Control Panel. Access to Control Panel groups can be done in a similar fashion. Restricting access to Control Panels in this way allows you to build Control Panels into hierarchical structures.

A special type of Control Panel access can be achieved by specifying which Control Panels are automatically opened when Data Explorer is started with the Image window or menu bar as the anchor (using the `-image` or `-menubar` option).

Creating, Modifying, and Deleting Control Panel Groups

The following describes how to create Control Panel groups, restrict access (build hierarchies), and specify which Control Panels are open at startup.

Control panel groups are created using the **`Control Panel Group...`** dialog box. The dialog box is opened by selecting the **`Control Panel Groups...`** option under the **`Options`** menu in the VPE window (Figure 64).

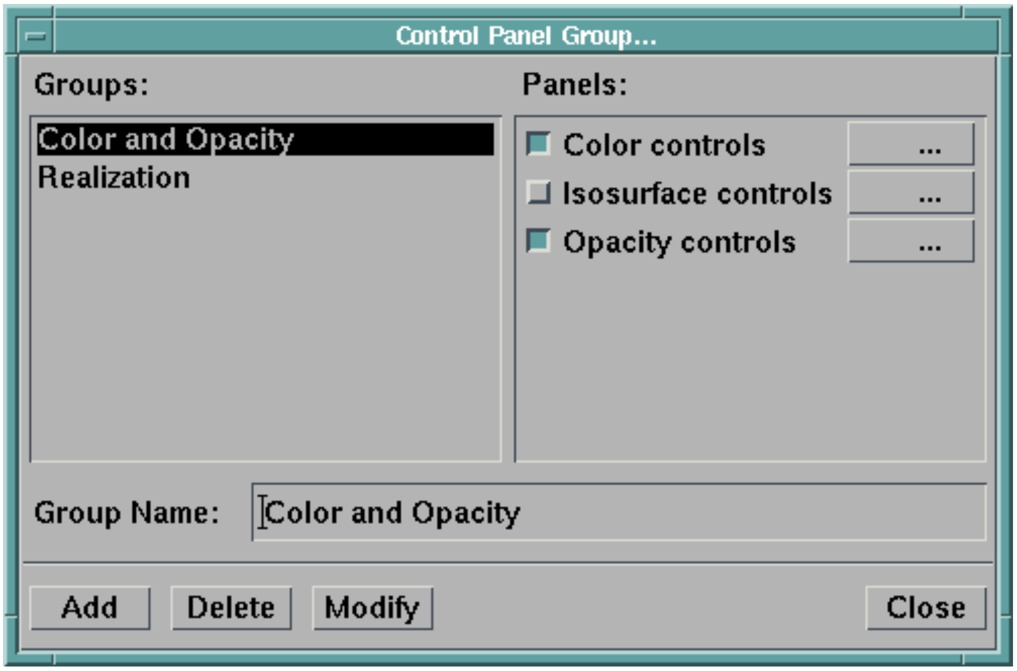


Figure 64. Control Panel Group Dialog Box

The right side of the dialog displays the list of Control Panels. Associated with each Control Panel is a toggle button to the left and an ellipsis toggle to the right. The left side of the dialog is a list of the existing groups. This list remains empty until a group is created. At the bottom of the dialog box is a series of pushbuttons that are used for creating, modifying, and deleting groups. Clicking on the ellipsis causes the corresponding Control Panel to be opened (or raised to the front if it is already opened). Clicking on the ellipsis when it is depressed closes the corresponding Control Panel and releases the toggle.

To create a Control Panel group:

1. Select the Control Panels that you want in the group by activating the corresponding left toggle button.
2. Enter the name of the group in the text field next to **`Group Name`** Although it is not required, select a name that is unique compared to the names of other Control Panels or other groups.
3. Click on the **`Add`** pushbutton at the bottom of the dialog box.

The new group is added to the list of groups displayed on the left side of the dialog box. The group is also added to the list of named Control Panels that is displayed by the **`Open Control Panel by Name`** option in the **`Windows`** menu.

Once a Control Panel group is created, it can be modified. Modifying a Control Panel group can include changing its name, adding new Control Panels, and removing Control Panels from the group.

To modify a Control Panel group:

1. Click on the Control Panel group name that is to be modified. This causes the name of the Control Panel group to be displayed in the group name text field and causes the toggle buttons of the Control Panels that are members of the group to be activated. All other Control Panel buttons are released.
2. You may now change the name of the group by editing the text field. Add a new Control Panel by clicking on its toggle button (this causes the toggle button to be activated). To remove a Control Panel from the group, click on its toggle button to release it.
3. Once you have made the desired changes to the Control Panel group, click on the **Modify** button at the bottom of the dialog box. This causes the change to take effect.

To delete a Control Panel group:

1. Select the group by clicking on its name.
2. Click on the **Delete** button at the bottom of the dialog box.

Restricting Control Panel Access

Access to Control Panels is restricted using the **Control Panel Access...** dialog box. This dialog box is opened by selecting the **Control Panel Access** in the **Option** menu of the VPE, Image, or Control Panel windows. The **Control Panel Access** dialog is used to restrict certain Control Panel names or Control Panel groups names from appearing in the **Open Control Panel by Name** option in the **Windows** window.

A **Control Panel Access** dialog box appears in Figure 65.

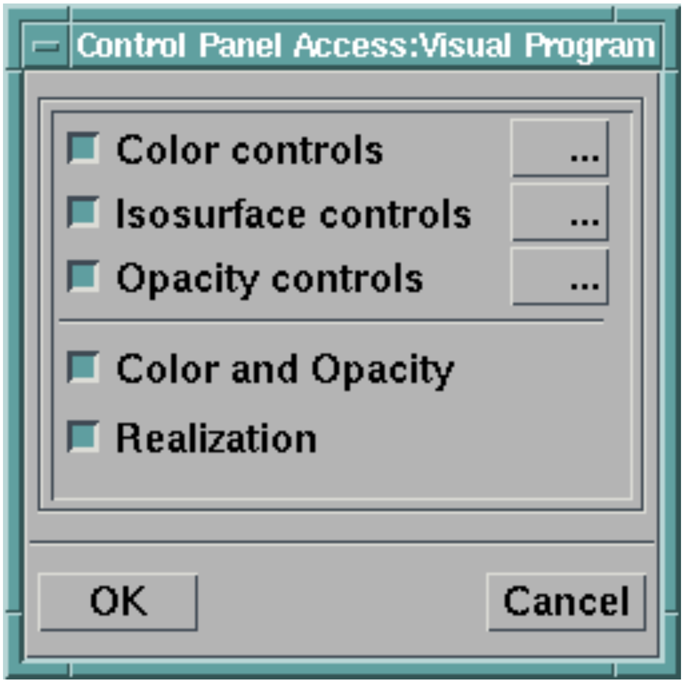


Figure 65. Control Panel Access Dialog Box

Each Control Panel and Control Panel group is listed. To the left of each is a toggle button. This toggle button is used to select which Control Panels or groups can be accessed from the window in which the dialog box was opened. To the right of the Control Panel names is an ellipsis toggle. (Note that Control Panel groups do not have the ellipsis.) The ellipsis toggle is used to open (or raise) the corresponding Control Panel when the button is activated. Once activated, selecting the button again closes the Control Panel and pops up the button.

To restrict Control Panel access:

1. Click on each Control Panel access toggle button that you wish to exclude so that those toggle buttons are deactivated. (Initially, all the toggle buttons are activated.)
2. Once you have indicated which Control Panels and Control Panel groups you wish to exclude, click on **OK**.

Selecting the **Cancel** pushbutton causes the dialog box to be closed and any changes to be canceled.

Specifying a Startup Control Panel

Data Explorer allows you to choose whether a Control Panel opens automatically when you start the system with the Image window or menu bar as the anchor (using the `-image` or `-menubar` option). Using this option, you can have the appropriate Control Panels be immediately available to a user running your visual program. The **Startup Control Panel** option in the **Options** pull-down menu is a toggle button, and is toggled on by default. If you do not want a particular Control Panel to open automatically, toggle the option off by clicking on it in that Control Panel.

The automatic startup feature can be suppressed by using the `-suppress` startup flag when you run Data Explorer. See C.2, "Command Line Options" on page 295 for more information.

Opening Existing Control Panels

You can open existing Control Panels associated with a visual program in the following ways:

- Select the **Open All Control Panels** option from the **Windows** menu in the VPE or Image window, or from the **Panels** menu in an already open Control Panel.
- Select the **Open Control Panel by Name** option from the **Windows** menu in the VPE or Image window, or from the **Panels** menu in an already open Control Panel. From the list of Control Panel names, click on the one you want to open. (If there are no accessible Control Panels, no names are displayed.)
- Double-click on the interactor stand-in in the VPE whose Control Panel you want to open. All Control Panels that contain that interactor are opened. If the selected interactor does not currently have a Control Panel, Data Explorer creates one for it.

  Alternatively, you can select one or more interactor stand-ins in the VPE, then choose **Open Control Panel** from the **Windows** menu in the VPE. All Control Panels associated with the selected interactors are opened.

When Data Explorer is started with the Image window or menu bar as the anchor window, some Control Panels may be opened automatically as a visual program is loaded. For visual programs you create, you can decide whether a Control Panel should open automatically (see “Specifying a Startup Control Panel”).

Using Interactors

You use interactors to dynamically change the inputs of a tool without making modifications in the VPE window. Interactors reside in Control Panel windows. As a visual program is built in the Editor window, the user selects the interactor stand-ins from the Tool Palettes and places them on the canvas. Then the corresponding interactors are placed into existing Control Panels or into a new Control Panel, as described in “Building Control Panels” on page 129. Different interactor stand-ins can be represented by different interactor styles.

Integer and Scalar Interactors

Integer and scalar interactor stand-ins can be represented by four styles:

- Stepper
- Dial
- Slider
- Text

For both integer and scalar stand-ins, the stepper is the default. You can change the style at any time by using the **Set Style** option from the **Edit** pull-down menu on the Control Panel.

Stepper: The Stepper (Figure 66) enables you to enter a value by typing it into the text field or by using the arrow buttons to increase (right arrow) or decrease (left arrow) a displayed value. The arrow buttons have a built-in acceleration function so that the longer you depress a button, the faster the value changes.

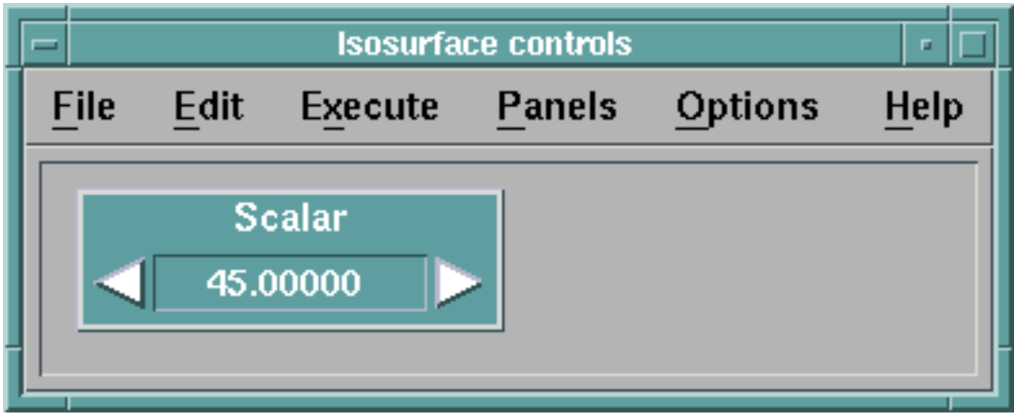


Figure 66. Stepper Style

Dial: The Dial (Figure 67 on page 143) has circular shape. You can specify a value by manipulating the dial indicator, or you can directly enter a value in the field at the bottom of the interactor.

To manipulate the dial indicator, press and hold the mouse button in the circular part of the interactor. Turning the dial clockwise increases the interactor value, while turning it counterclockwise decreases the value. If you move the cursor within the dial, shading occurs in intervals around the dial indicator. The shading indicates whether the value change is positive or negative. The shading is light if you move in a clockwise direction (positive); it is dark (negative) if the movement is counterclockwise. If you click the pointer within the shaded area, the dial indicator jumps to the mouse pointer location, and the value changes accordingly.

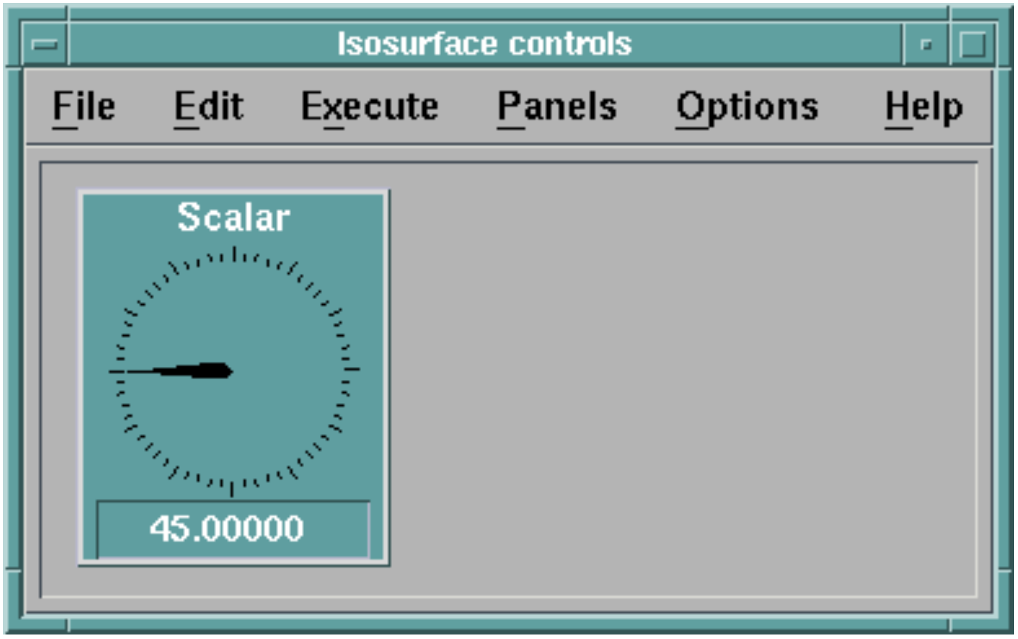


Figure 67. Dial Style

The dial indicator can move clockwise or counterclockwise as many times as determined by the increment values and minimum and maximum that are set. These limits can be set in the Set Attributes dialog box, which you can display by selecting the **Set Attributes...** option on the **Options** menu. When the interactor reaches the limit, it can no longer be turned in that direction.

Slider: The Slider (Figure 68) enables you to enter a value by either moving the tab on the Slider, typing in a number, or clicking on one of the arrow buttons to increase (right arrow) or decrease (left arrow) a displayed value.

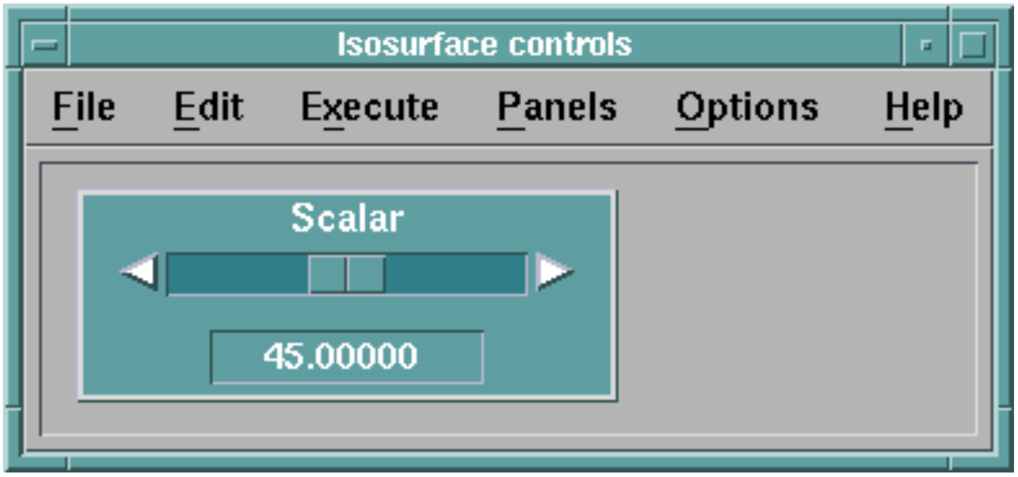


Figure 68. Slider Style

Text: The Text style (Figure 69 on page 144) enables you to simply type in a value. For more information on how to enter and modify text in a field, see “Editing Text Fields” on page 64.

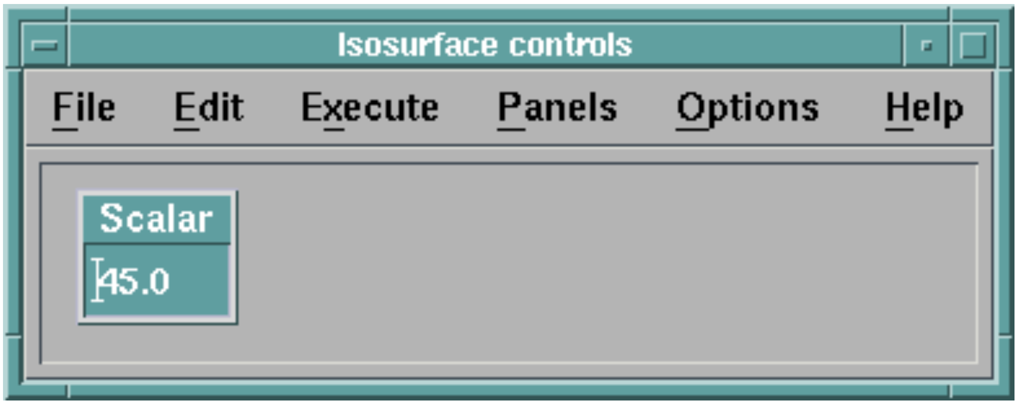


Figure 69. Text Style

String Interactor

The string stand-in has one style of interactor, which cannot be changed. This interactor consists of a text field (Figure 70). (For information on how to enter and modify text in a text field, see “Editing Text Fields” on page 64.) It enables you to enter strings by typing directly into the text field, then pressing the Enter key.

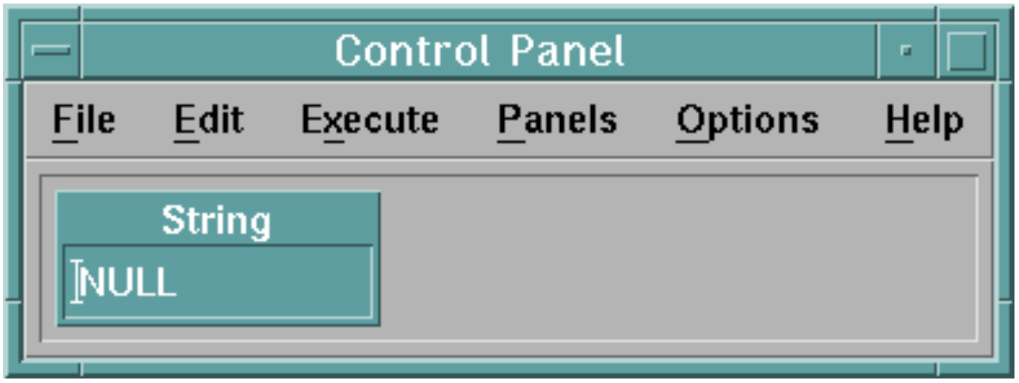


Figure 70. String Interactor

Value Interactor

The value stand-in has one style of interactor that cannot be changed. This interactor consists of a text field. (For information on how to enter and modify text in a text field, see “Editing Text Fields” on page 64.) This interactor enables you to enter scalars, vectors, and lists by typing directly into the text field and pressing the Enter key. The input into the text field must either begin with a numeric value, or be enclosed by brackets ([ ]) for vectors. See “Vectors, Matrices, and Tensors” on page 195 and “Lists” on page 196 for more information on the syntax of vectors and lists.

Vector Interactor

The vector stand-in has two styles of interactor; a stepper style and a text style. The Vector interactor works like one, two, or three Steppers stacked in a column. With it, you can specify the components of a vector (e.g, x, y, and z) from top to bottom. See “Stepper” on page 142 for detailed information on the operation of a Stepper. You can also change the dimensionality of the interactor using the **Edit** menu **Set Dimensionality** option. The text style vector interactor is similar to that of the value interactor.

List Interactors

Data Explorer provides six types of list stand-ins and interactors:

- Integer
- Scalar
- Selector
- Vector
- String
- Value

With the exception of Selector, these list interactors can have two styles: list-editor and text. The text-style list interactor is similar to that of the value interactor.

The list-editor style list interactors consist of the following parts:

- Title
- List of values
- Pushbuttons for adding and deleting elements
- A single stepper (for integer and scalar lists), three steppers (for a vector list), or a text field (for string and value lists).

Figure 71 shows an example of a Vector list interactor. For the SelectorList interactor, see "Selector and SelectorList Interactors" on page 146.

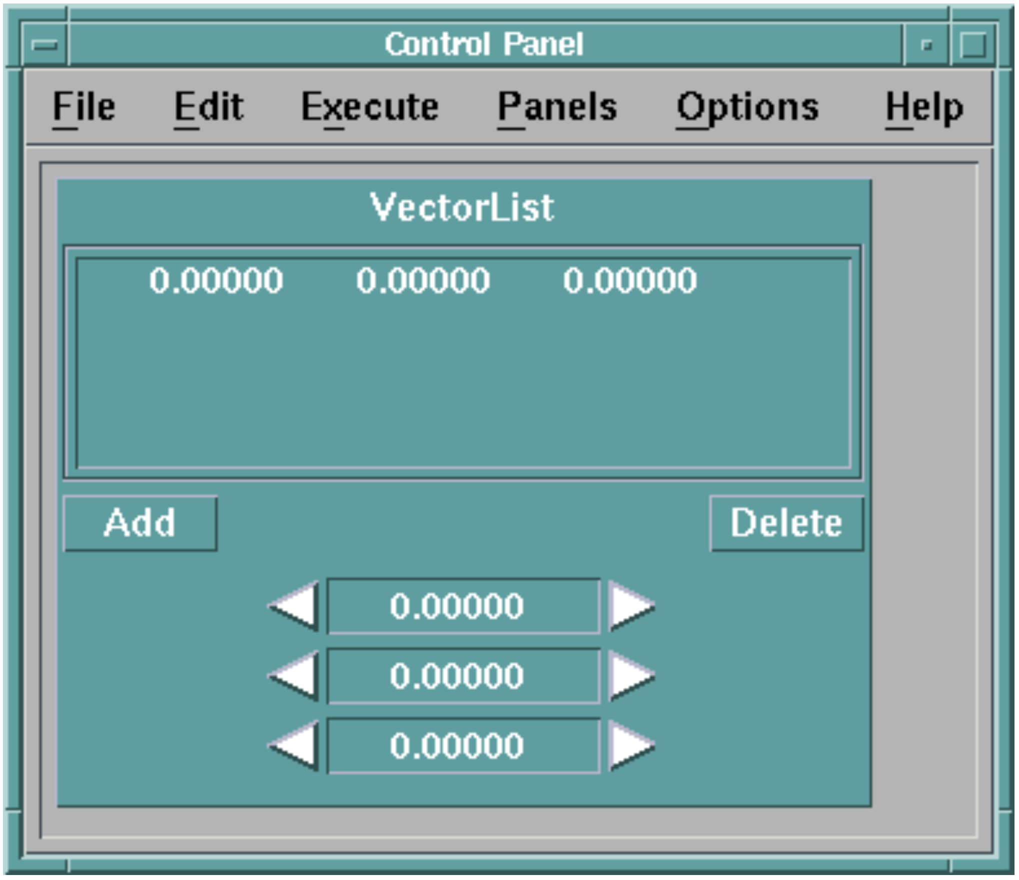


Figure 71. Sample Vector List Interactor

The top portion of the list interactor shows the current list of values. If the list exceeds the length of the display area, a vertical scroll bar is provided. If the list values exceed the width of the display, a horizontal scroll bar is provided.

Modifying an Element in the List: To modify an element, select it by clicking on it. The stepper or text field is updated to show the element's values. Use the stepper or text field to change the values of the elements.

Appending an Element to the List: To append a value to a list, make sure that no list elements are selected. (An element can be deselected by clicking on it.) Use the steppers or text field at the bottom of the interactor to specify the value, then click on the **Add** button.

Adding an Element to the Middle of the List: To add an element to the middle of the list, select the element following the position you want the new value to occupy. Click on the **Add** button. A copy of the selected element is added to the list, becomes selected, and the steppers or text field display its current value. Use the steppers or text field to adjust the value of the new element.

Deleting an Element from the List: To delete an element from the list, select it by clicking on it, then click on the **Delete** button. After deleting the item, the next item in the list becomes selected.

Selector and SelectorList Interactors

The Selector interactor (Figure 72.) can be used as a switch control in a visual program. It can appear as an option menu (with only the current choice shown) or as a "radio button" interactor, with all possible choices shown, and only the current choice highlighted in the radio button next to the label. The SelectorList interactor always appears as a list of toggle buttons.

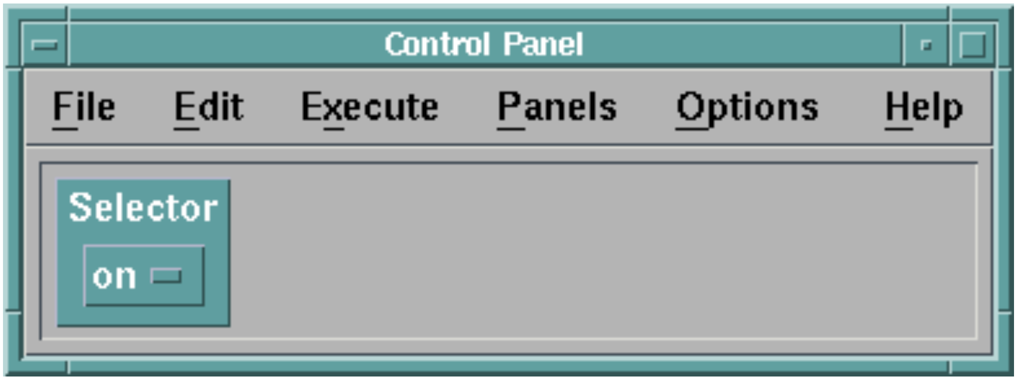


Figure 72. Selector Interactor (Radio-button Style)

"Setting Selector and SelectorList Interactor Attributes" on page 136 describes how to configure the selector interactor for your visual program, and also describes its various uses.

FileSelector Interactor

The FileSelector interactor can be used to select a file from within the file system (Figure 73 on page 147).

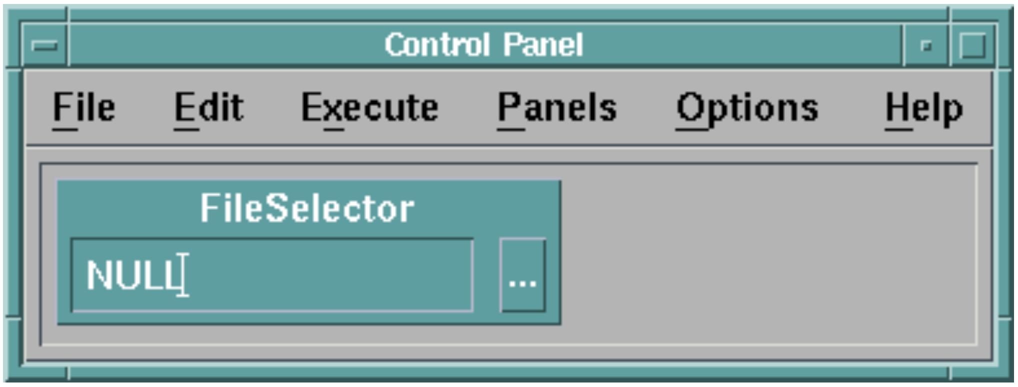


Figure 73. FileSelector Interactor

The interactor consists of a text field containing a string and a button labelled with an ellipses. Clicking on the button causes a file selection dialog box to be opened. The file selection dialog box, illustrated in Figure 74 on page 148, functions in a similar manner to other file selection dialog boxes (for a description on how to use file selection dialog boxes see “Saving and Restoring a Visual Program” on page 115) with the exception of the buttons at the bottom of the dialog box.

OK causes the file name in the Selection area to be set in the text field of the FileSelector interactor and closes the file selection dialog box. **Filter** applies the filter string specified in the Filter area. **Close** closes the file selection dialog box without any modification to the FileSelector interactor text field. **Apply** causes the file name in the Selection area to be set in the text field of the FileSelector interactor and leaves the file selection dialog box open.

An alternate way of specifying or modifying a file name in the FileSelector text field is to directly type into the field. See “Editing Text Fields” on page 64 on how to enter and modify text in a text field.

The FileSelector interactor produces two outputs: the first output is the contents of the text field (typically a fully qualified path name set from the file selection dialog box). The second output is the name of the file as it appears in the directory (that is, excluding any directory name).

Reset Interactor

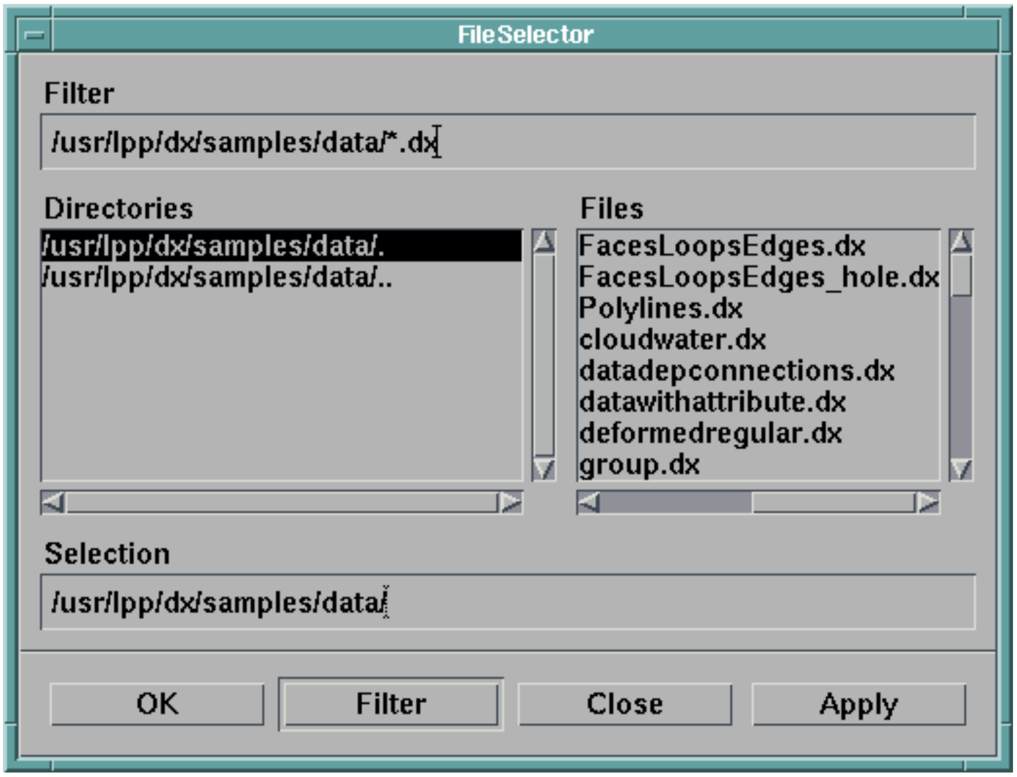
The Reset interactor outputs one value for the first execution after its toggle is set, and a different value thereafter. This interactor appears only as a toggle button.

Toggle Interactor

The Toggle interactor outputs one of two possible values. The values can be strings, scalars, vectors, or matrices. This interactor appears only as a toggle button.

Using Data-Driven Interactors

Most of the interactor types may be *data-driven*, meaning that their attributes, such as minimum, maximum, increment, and label, may be set by connecting the output of a tool to the input of the interactor stand-in in the VPE or by a value typed into the interactor stand-in’s Configuration dialog box, rather than by using the **Set Attributes** dialog box for the interactor.



Filter: /usr/lpp/dx/samples/data/*.dx

Directories:
/usr/lpp/dx/samples/data/.
/usr/lpp/dx/samples/data/..

Files:
FacesLoopsEdges.dx
FacesLoopsEdges_hole.dx
Polylines.dx
cloudwater.dx
datadepconnections.dx
datawithattribute.dx
deformedregular.dx
group.dx

Selection: /usr/lpp/dx/samples/data/

Figure 74. File Selection Dialog Box

If an interactor is data-driven, then the information transmitted via connections or set in the Configuration dialog box overrides the values set via the **Set Attributes** dialog box and causes the corresponding values in the **Set Attributes** dialog box to be grayed out.

Data-driven interactors allow you to create visual programs that will work with a variety of input data sets without the need to reset the interactor attributes to be in a range appropriate for the data being used. For example, a scalar interactor controlling an isosurface value can be data-driven by connecting the input data field to it. The interactor is then automatically set so that its minimum and maximum span the range of the data.

Data-driven interactors have a data input to which an input data field may be connected. In this case the interactor automatically chooses the minimum, maximum, and increment. However, if you would like to have more control over the exact values that are used, the interactors allow you to specify them directly through other input tabs that are by default hidden. For example, you may wish to set the minimum and maximum for an interactor to go from the minimum of the data values to the midpoint of the data values, rather than to the maximum. In this case, you can use the “min” and “max” input tabs of the interactor rather than the “data” tab.

The interactors that can be data-driven are Integer, Scalar, Vector, IntegerList, ScalarList, VectorList, Selector, SelectorList, and Toggle. In Chapter 2, Functional Modules in *IBM Visualization Data Explorer User’s Reference*, the inputs for each of these interactors are described on the manual page corresponding to that interactor.

Each time an input to a data-driven interactor is changed (for example, by importing a new data set) the interactor is reexecuted, updating its attributes. If the current setting of the interactor lies within the new range allowed, the interactor value does not change. If the current setting is outside the new allowed range, the current setting is reset to the midpoint of the new minimum and maximum.

7.2 Creating and Using Macros

In general, macros are higher level processing functions that are constructed from simpler ones. A Data Explorer *macro* is a sequence of modules collected together that you can use as a single tool. This is useful if you have a sequence of modules that provide a function that you need frequently in your visual programs. Macros can help make your visual programs simpler by combining several tool icons into one.

Creating Macros

Macros are themselves visual programs and are created in much the same way. For example, the macro represented in Figure 75 on page 150 maps data onto a plane, colors it, and then performs a deformation using the RubberSheet module. It is constructed by placing the appropriate tool icons on the VPE canvas and connecting them.

Note: You can also create a macro directly from a visual program in the VPE window. See “Creating Macros the Easy Way” on page 152.

Macro’s inputs and outputs are represented by the **Input** and **Output** tools from the **Special** category. A macro has at least one input or output.

Note: Macros cannot include interactors, Sequencers, colormaps, probes, picks, Display, or Image tools.

Place the **Input** icon on the canvas and connect it to the appropriate tool icon input tab. The **Input** type conforms to that of the input tab to which it is connected. Like other tools, the **Input** icon can be connected to multiple input tabs. You can use the Configuration dialog box for the **Input** tool (Figure 76 on page 151) to specify:

- Tab position in the macro
- Name of the input
- Default value for the input
- Description of the input
- Whether the input is a required parameter
- Whether the input has a descriptive default value.

Some of these values are, by default, inherited from the input tab to which the **Input** tool is connected, but many of them can be changed from the Configuration dialog box. The fields in the Configuration dialog box for the **Input** tool are:

Position Specifies the tab position this input occupies on the resulting macro icon. By default, the tab positions are assigned in the order that you select the **Input** icons from the tool palette. For example, the first icon you drag over to the canvas has position 1, the second has position 2, and so on.

If an input is deleted from a macro, the position of the other inputs are *not* automatically changed. For example, if the input at position 2 of a macro with three inputs is deleted, the other two inputs

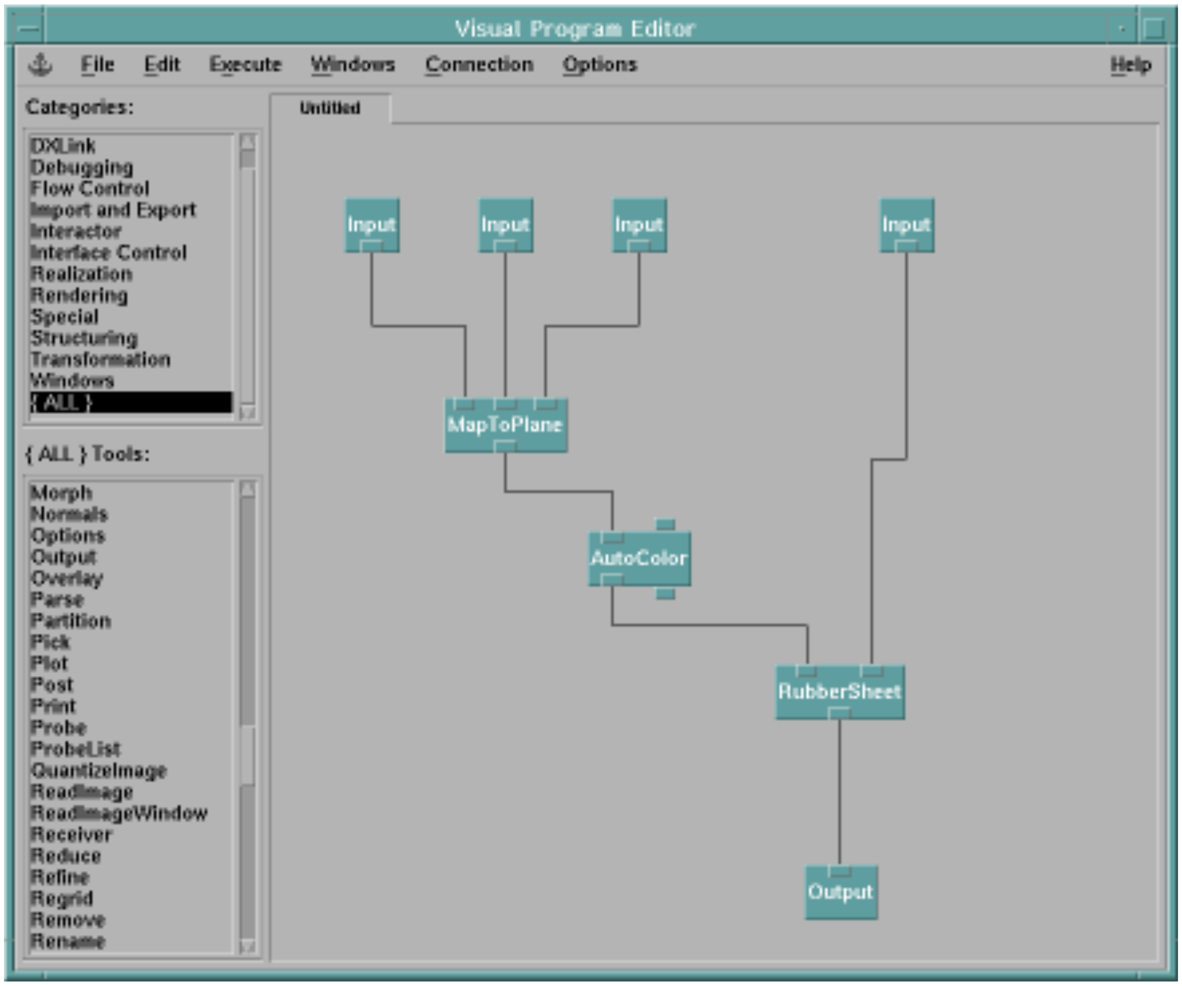


Figure 75. Example of a Macro

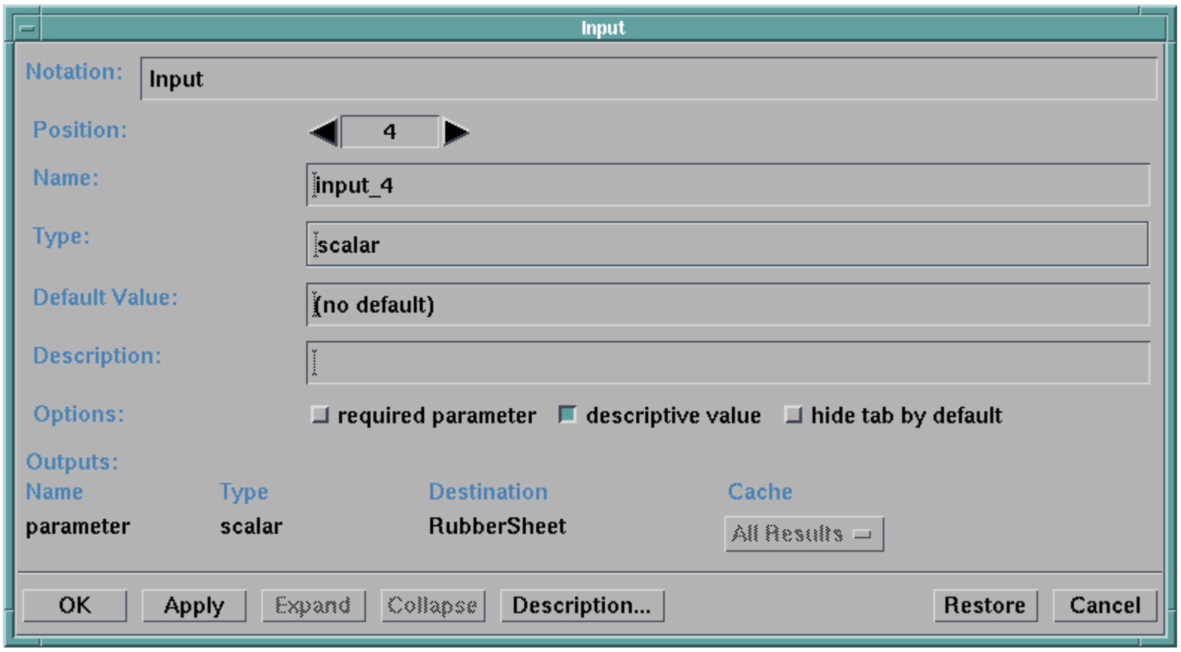
remain at positions 1 and 3, with position 2 left empty. Each input *must* have a different value for position.

Name Specifies the name of the input. This field must begin with a letter, and no spaces or special characters, except for the at-symbol (@) and the underscore character (_), are allowed.

Type Specifies the type of the input. The type conforms to the inputs that the **Input** tool is connected to. When connecting it to more than one input, the **Input** tool assumes the type of the most restrictive input to which it is connected. For example, if the **Input** tool is connected to two inputs, an object and a scalar, the **Input** tool type would become a scalar type.

Default Value Specifies a default value for the input. If the **descriptive value** toggle button is activated, the default value field of the macro is treated as a description, not an actual value. You can use this to indicate to users of the macro what sort of parameter this input should be.

Description Provides a short description of the input for your own documentation purposes. This description is used to generate the Description window for the resulting macro tool, which you can access through its Configuration dialog box.


Input

Notation: Input

Position: 4

Name: input_4

Type: scalar

Default Value: (no default)

Description:

Options: ☐ required parameter ☑ descriptive value ☐ hide tab by default

Outputs:

Name	Type	Destination	Cache
parameter	scalar	RubberSheet	All Results

Figure 76. Input Configuration Dialog Box

Options The **required parameter** toggle button, when activated, indicates that the input is required for the macro. The corresponding input tab of resulting macro icon is highlighted with a different color. This option is inherited from the input to which the **Input** tool is connected.

The **descriptive value** toggle button is used with the **Default Value** field, as described above.

Specify the macro's outputs with the **Output** tool from the **Special** category. Place the **Output** tool icon on the canvas and connect it to the appropriate output tab from another tool icon. The **Output** also inherits its type from the output tab to which it is connected.

The Configuration dialog box for the **Output** tool is similar to the one for **Input**, but the **Options** toggle buttons and the **Default Value** field are grayed-out.

Before you can use the macro, you must first name it, save it, and then load it, making it available in the tool palette.

To name the macro, select the **Macro Name** option from the **Edit** pull-down menu. The **Name...** dialog box opens. The **Name** dialog box for the example macro is illustrated in Figure 77 on page 152. Type a name for the macro in the **Name** field. The name must consist only of letters and numbers (no spaces or underscores), and must begin with a letter. By default, the macro is assigned to the "Macros" category. You can assign the macro to a different category by typing a new name in the **Category** field. The category you choose can be a new or existing category.

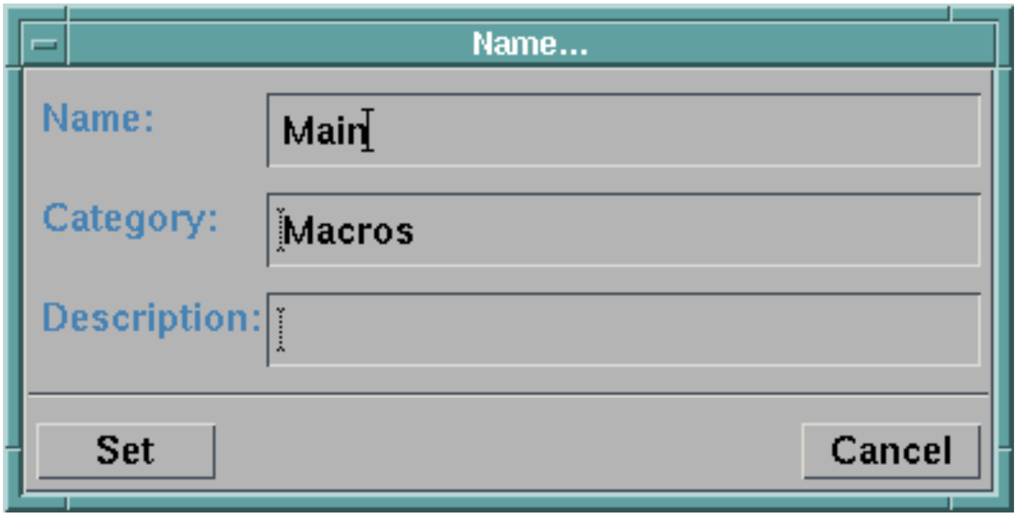


Figure 77. Macro Name Dialog Box

To save the macro after it has been named, use the **Save As...** option of the **File** pull-down menu. Because macro files are saved as visual programs, and therefore have **.net** extensions, it may be helpful to organize them in a separate directory to distinguish them from other visual programs.

Creating Macros the Easy Way

For use in other visual programs or to simplify the "network" that appears in the VPE canvas, you may want to combine some of the tools of a visual program in a single macro. This can be done directly and easily in the VPE window if the selection includes only allowed tools (see preceding **Note**, page 149).

1. On the VPE canvas, use the left mouse button to create a selection box around the tool icons to be included in the macro. (It may be necessary to first reposition some of the icons in order to simplify this "boxing" process.)

   Note: Alternatively, you can "shift click" on all the desired tools.

2. Select **Create Macro** from the **Edit** pull-down menu.
3. In the **Create Macro** dialog box that appears, enter the required information: macro name, category, description, and file or path name. The default category ("macro") and description ("new macro") can be changed or left as they are.
4. Click on **OK**. Unless you specified otherwise, the macro will be saved in the directory from which Data Explorer was started, under the name you specified (input and output names can be changed by editing this saved file) using the Visual Program Editor. The new macro will appear in the category specified, under the name specified in the dialog box.

Loading Macros

For a macro to be available for use, it must be loaded into the tool palette. To load a saved macro:

1. Select the **Load Macro...** option from the **File** pull-down menu in the VPE. A file selection dialog box appears. For information about how to use the dialog box, see "File Selection Dialog Boxes" on page 116.

2. Use the dialog box to locate the macro you want to load, and select the desired file. The macro name appears on the tool palette under its assigned category.
3. You can also load all the macros in a selected directory by clicking on **`Load All Macros`** in the dialog box.

   Note: If the macro you loaded has the same name as a macro previously loaded, the "new" macro replaces the earlier one, whether or not they were assigned to the same category.

When the new macro is available for use, it is listed in the tool palette as a member of the category you assigned it to. For example, the MapAndDeform macro would be listed under the MyMacros category.

When you save a visual program which uses a macro, the path to the macro is saved in the visual program. When you reload the visual program, the macro will automatically be loaded for you from the same location, so it will only be necessary to explicitly load the macro if it has been moved to a new location. You can also easily see what macros need to be included with a visual program, if for example, you are sending a visualization program to another user. Simply look at the top of the .net file for the list of referenced macros.

You can configure Data Explorer to automatically load macros from a specified directory when you begin the Data Explorer program. To do this, use the `-macros` *`pathlist`* option when starting Data Explorer, where *`pathlist`* specifies one or more paths where the macros can be found. Alternatively, you can use the *DXMACROS* environment variable. For more information on how to do this, see "Using Environment Variables" on page 59, and C.1, "Environment Variables" on page 292.

Using Macros in a Visual Program

After you have created and loaded a macro, you can use it the same way you use other Data Explorer tools. Simply select it from the tool palette and place the icon on the canvas. Figure 78 on page 154 shows the new icon created for the MapAndDeform macro.

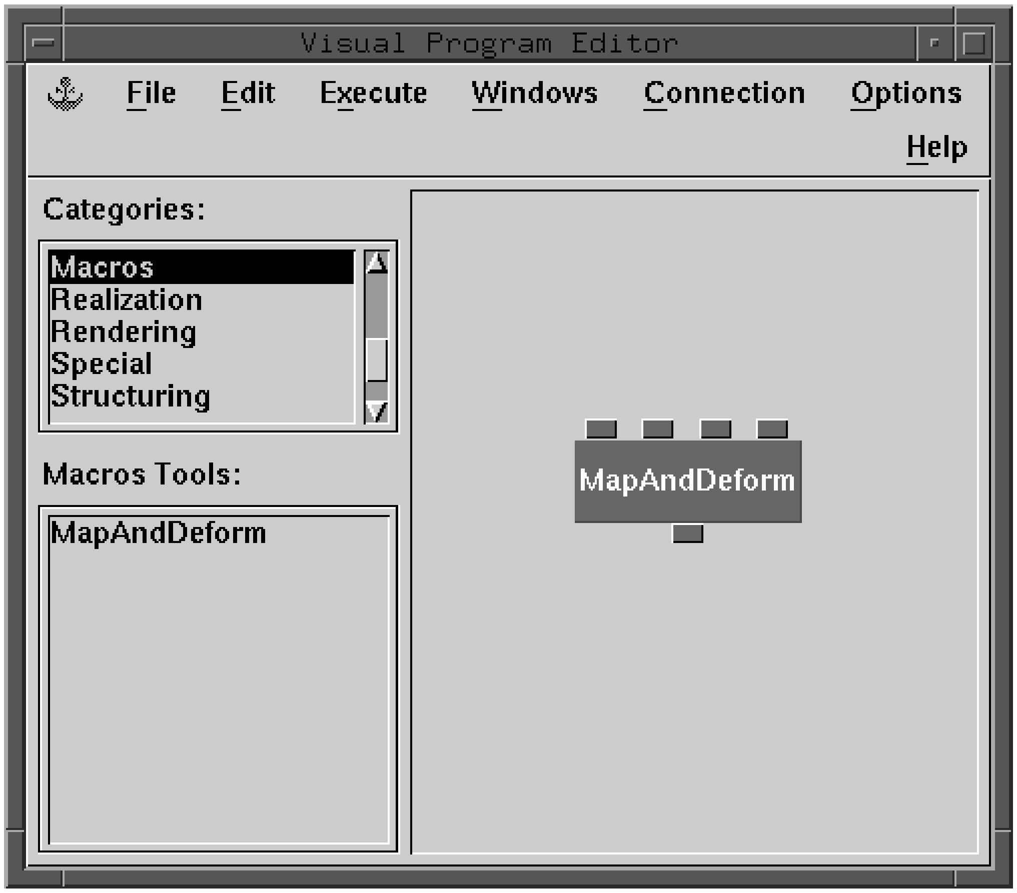


Figure 78. MapAndDeform Macro Icon

The macro has a Configuration dialog box describing the input and output parameters. You can open the dialog box by double-clicking on the macro icon.

Viewing and Changing Macros

After you place a macro in a visual program, it is possible to view the macro contents and to change some of the tools inside the macro.

To open a macro:

1. Select the macro icon by clicking on it.
2. Select the **Open Selected Macro(s)** option from the **Windows** pull-down menu. A VPE window is opened, with the expanded macro representation.

You can repeat these steps for any macros contained in the newly opened VPE, until only modules are present in the window.

With the macro opened, you can change the tools inside the macro. However, you cannot change the number or ordering of inputs or of outputs.

Changing the tool in a macro affects only the current visualization session. If you want the changes to be permanent, you must save the changed macro, using the **Save** or **Save As...** options. If you want to restore the definition of the macro to the way it was before you modified it, close the VPE window containing the macro (without saving) and reload the macro from the main VPE window.

Chapter 8. Graphical User Interface: Menus, Options, and the Message Window

8.1 Using the Primary Window Pull-Down Menus and Options

This section provides illustrations of the primary windows with explanations of the pull-down menu options on their menu bars. The discussions include:

- “VPE Window Menu Bar”
- “Control Panel Menu Bar” on page 162
- “Image Window Menu Bar” on page 165
- “Colormap Menu Bar” on page 168
- “Menu Bar Menu Bar” on page 170
- “Message Window Menu Bar” on page 172

The Sequencer and Help primary windows have no pull-down options, so they are not discussed here. See “Using the Sequencer” on page 68 for information about the Sequencer, and 5.3, “Using Online Help” on page 65 for information on the Help window and Help menu options.

VPE Window Menu Bar

The menu bar displays the titles of seven menus:

- File
- Edit
- Execute
- Windows
- Connection
- Options
- Help.

The following sections describe the menus and their options. For many of the menu options, one or more tools on the canvas must be selected in order for the option to be applied. If a menu option is grayed out, it is unavailable and cannot be selected.

VPE File Menu

The **File** menu displays the options you can use to create new programs, view or edit previously created programs, load macros, and save programs. You must have write permission for a file to save changes directly to that file.

The following options appear on the **File** menu:

New Initializes the VPE for a new program and clears the canvas. If you have opened Control Panels and Image windows from the VPE, they are closed. If a program you changed but did not save currently exists on the canvas, a dialog box appears so you can save changes. If you opened the VPE using **Open Macro** or **Open Visual Program Editor**, this option is grayed out.

Open Program...
Opens an existing visual program. A dialog box appears for you to specify the desired program name. If a program you changed but did not save currently exists on the canvas, a dialog box appears asking if you want to save changes. If you opened the VPE using **Open Macro**, this option is grayed out. See “File Selection Dialog Boxes” on page 116 for more information.

Save Program

Saves the current visual program, and associated Control Panels. This option saves the following files: *filename*`.net` (for network; that is, a Data Explorer visual program); *filename*`.cfg` (for configuration).

Save Program As...

Saves the current visual program and all associated Control Panels under a name you type into the dialog box. This option saves the following files: *filename*`.net` (for network; that is, a Data Explorer visual program), *filename*`.cfg` (for configuration).

Program Settings

Brings up a cascade menu with two choices:

- `Save As` saves the current settings for the visual program (e.g., window placement, interaction mode, camera viewpoint, interactor setup, etc. The settings are saved in a `.cfg` file.
- `Load` loads previously saved settings from a `.cfg` file.

Load Macro...

Loads a macro and makes it available in a tools palette. A file selection dialog box appears for you to specify the desired macro file name. See also “Loading Macros” on page 152

Load Module Definition(s)...

Loads a module definition and makes it available in a tool palette. A file selection dialog box appears for you to specify the desired module file name. The executable you are using must either include this module, or this module definition must refer to an outboard or run-time loadable module. See 9.2, “Loading and Using Outboard and Runtime-Loadable Modules” on page 183.

Print Program...

Prints or saves to a file a representation of a visual program.

If the `Label Set Input` toggle button is activated, then any parameter value(s) set in the configuration module’s Configuration dialog box will also be printed next to the corresponding tab(s), if scale allows.

Quit or **Close**

Closes all windows created from the current VPE window. If you have made changes since the last save, a dialog box appears so you can save changes. If you opened the VPE from the Image window or with the `Open Macro` option, `Close` appears as an option instead of `Quit`. `Quit` is grayed out if the `Execute` menu title is highlighted. You must select the `End Execution` option in the `Execute` pull-down menu; or if the Sequencer is running, you must stop it or pause before quitting.

VPE Edit Menu

The `Edit` menu lists options for manipulating tools on the canvas.

Configuration...

Displays a Configuration dialog box for setting and changing the values of the selected tool. See “Entering Values in a Configuration Dialog Box” on page 107.

Find tool...

Opens a dialog box that lists the tools in the visual network displayed on the canvas. Double-click on the tool name (or type the name in the

`Selection` field) to highlight the corresponding icon in the canvas. If the icon falls outside the canvas, the canvas will automatically scroll to display the part of the network in which it lies.

See "Locating Tools: The Find Tool Dialog Box" on page 111.

Input/Output Tabs
Brings up a cascade menu with the following options:

Add Input Tab	Adds input tabs to the selected tool.
Remove Input Tab	Removes input tabs from the selected tool.
Add Output Tab	Adds output tabs to the selected tool.
Remove Output Tab	Removes output tabs from the selected tool.
Reveal All Tabs	Reveals all tabs of the selected tool.
Hide All Tabs	Hides all unconnected tabs of the selected tool.

Assign Get/Set Scope
Brings up a cascade menu with the following options:

Convert All Modules	Brings up a dialog which allows you to interactively change all of the Get and Set tools in your visual program to either the Local or Global variety.
Set Selected Gets/Sets Local	Sets all of the selected Get and Set modules to the Local variety
Set Selected Gets/Sets Global	Sets all of the selected Get and Set modules to the Global variety

For additional information on the Get/Set tools see 4.6, "Preserving Explicit State" on page 45.

Select/Deselect Tools
Brings up a cascade menu with the following options:

Select All	Selects all tools on the canvas.
Select Connected	Selects all tools connected to the currently selected tool.
Select Unconnected	Selects all tools not connected to the currently selected tool.
Select Upward	Selects all tools (modules) that directly or indirectly provide input to the selected tool.
Select Downward	Selects all tools (modules) that directly or indirectly accept output from the currently selected tool.
Deselect All	Deselects all selected tools.
Select Unselected	Selects all unselected tools.

Output Cacheability

Brings up a cascade menu with the following options:

Optimize Cacheability

Uses a heuristic for automatically selecting the optimal cache setting for each module in a visual program.

Set Output Cacheability

Brings up a cascade menu with the following options:

Cache All Results	Caches all results of the selected tool(s)
Cache Last Result	Caches only the last result of the selected tool(s)
Cache No Results	Caches no results.

Show Output Cacheability

Brings up a cascade menu with the following options:

All Results	Highlights all tools for which all results are cached
Last Result	Highlights all tools for which only the last result is cached
No Results	Highlights all tools for which no results are cached

See “Cache Control: Executive” on page 215.

Delete

Deletes the selected tools and connections to other tools.

Cut

Deletes the selected tools but places them into a buffer so they can be pasted.

Copy

Copies the selected tools and places them into a buffer so they can be pasted.

Paste

Pastes the tools from the buffer into a selected location

Add Annotation

Adds annotation text to the canvas (see “Adding Annotation to a Visual Program” on page 115).

Insert Visual Program

Brings up a file selection dialog box for selecting a visual program. The selected program will be inserted to the right of all the tools in the current program.

Create Macro

Creates a macro from the currently selected tools. See “Creating Macros” on page 149 for more information.

Page

Brings up a cascade menu with the following options:

Create Empty Page	creates an empty page
Create with Selected Tools	creates a new page containing the currently selected tools
Delete	deletes the currently displayed page (available only if there is more than one page)
Configure	brings up a dialog which allows you to specify the name of the page, specify whether or not the page is included in PostScript output (through the Print Program option of the File menu), and specify the tab position of the page.

Macro Name ...

Displays a Configuration dialog box for naming a visual program as a macro. See “Creating Macros” on page 149 for more information.

Execution Groups...

Displays a dialog box for creating and modifying execution groups. (See 9.1, “Using Distributed Computation” on page 178.)

Comment ...

Displays a dialog box for documenting the function of a visual program or macro. (See “Adding Comments to a Visual Program” on page 114.)

VPE Execute Menu

You can use the **Execute** menu to execute the current visual program in various ways. This menu is the same as the **Execute** menus in control panels, the Image window, the Colormap Editor, and the Message window. The options in the **Execute** pull-down menu are grayed out when there is no connection to the server.

The Execute menu lists the following options:

Execute Once

Causes the visual program to execute only once, using current values and configuration settings. Subsequent interactor value changes do not cause program reevaluation until this command is reexecuted. This option is grayed out if the Sequencer is running.

Execute on Change

Causes the visual program to execute each time a value or configuration setting is changed. If you are changing values faster than Data Explorer can generate images, the program executes as fast as possible, always using the current settings at the time an execution cycle begins. This option is grayed out if the Sequencer is running.

End Execution

Causes the visual program to stop executing.

Sequencer

Causes the Sequencer to display. This option is grayed out if there is no Sequencer in the visual program. While the Sequencer is running, you can change interactors, and those changes are reflected in subsequent frames. You can pause the Sequencer on a particular frame and explore that frame using the **Execute Once** and **Execute on Change** options.

"Using the Sequencer" on page 68 discusses the Sequencer in detail.

While the visual program is executing, the **Execute** label in the menu bar is highlighted and remains so until execution is completed. If **Execute on Change** is selected, the **Execute** label in the menu bar is highlighted with one color during execution, and another color the rest of the time.

For more information, see 5.4, "Executing a Visual Program" on page 67.

VPE Windows Menu

The **Windows** menu allows a user to open and create control panels, open previously defined macros included in the visual program, open the Image window, and open the Colormap Editor.

These options appear on the **Windows** menu:

New Control Panel
Opens a new Control Panel. If you have selected one or more interactor stand-ins on the VPE canvas, the new Control Panel contains those interactors. If you have not selected anything on the canvas, the panel is empty.

Open Selected Control Panel(s)
Opens all Control Panels containing selected (highlighted) interactor stand-ins. Control Panels that are already open are unaffected.

Open All Control Panels
Opens all the Control Panels for the current visual program.

Open Control Panel by Name
Opens a cascade menu with the names of the existing control panels. Using this, you can open a particular control panel.

Open Selected Macro(s)
Opens a VPE window containing the expanded representation of the selected macro icon. From this newly opened window, you can open the macros contained in the currently opened macro. You can repeat this operation until only modules are contained in the window. See "Using Macros in a Visual Program" on page 153 for more information. This options is grayed out if no macro is selected.

Open Selected Image Window(s)
Opens an Image window for each selected Display or Image tool icon in the current visual program. This option is grayed out if no Display or Image icons are selected.

Open Selected Colormap Editor(s)
Opens a window where you can map colors and opacities to data values, the results of which are displayed in the visual image. This option is grayed out if a Colormap icon is not selected. Features of the Colormap Editor allow you to:

- Control the range of data values over which the mapping occurs.
- Select the colors and opacities that are mapped to that range of values.

The Colormap Editor is discussed in detail in 6.3, "Using the Colormap Editor" on page 119.

Open Message Window
Opens a window where you can access error and warning messages and working information about your execution. See 8.2, “Using the Message Window” on page 174.

VPE Connection Menu

The `Connection` menu lists the following options:

Start Server Connects the user workstation to the server. For more information, see 9.3, “Connecting to the Server” on page 183.

Disconnect from Server
Disconnects the user workstation from the server.

Reset Server Flushes the cache, forcing the visual program to reaccess the data on the server the next time it executes.

See “Resetting the Server” on page 185.

Execution Group Assignment...
Displays a dialog box that allows you to assign execution groups to specific machine host names.

See “Assigning Execution Groups to Workstations” on page 181.

VPE Options Menu

The `Options` menu allows you to customize tools and window features.

The `Options` menu lists the following options:

Tool Palettes
Toggles between displaying the tool palettes and not displaying them. You can also use the `Ctrl+T` accelerator key.

Control Panel Access...
Allows you to specify which controls panels and control panel groups are accessible from the VPE. (See “Control Panel Access, Groups, and Hierarchies” on page 138.)

Control Panel Groups...
Displays a dialog box that allows you to create, modify and delete control panel groups. (See “Control Panel Access, Groups, and Hierarchies” on page 138.)

Grid ... Allows you to select the grid type you want to use on the canvas. A dialog box displays the choices. See “Customizing the VPE Window” on page 113 for more information about the `Grid...` option.

Help Menu

The options in this menu are discussed in 5.3, “Using Online Help” on page 65.

Control Panel Menu Bar

The Control Panel menu bar (Figure 60 on page 129) displays the names of six menus you can use within the Control Panel window:

- File
- Edit
- Execute
- Panels

- Options
- Help.

The following sections describe the options available on these menus.

Control Panel File Menu

The **File** menu lists the following options:

Close Closes the current Control Panel.

Close All Control Panels
Closes all control panels.

Control Panel Edit Menu

The **Edit** menu lists the options you can use to edit and delete interactors in the Control Panel.

Set Style...
Allows you to change the type of selected interactors on the layout area. See "Changing the Interactor Style" on page 134.

Set Dimensionality...
Allows you to set the dimensionality of the selected interactors in the layout area. See "Changing the Interactor Dimensionality" on page 134.

Set Layout...
Allows you to set the layout of the selected interactors in the layout area. You can specify horizontal or vertical. See "Changing the Interactor Layout" on page 134.

Set Attributes...
for integer, scalar, and vector values, allows you to specify:

- The minimum and maximum limits for the interactor values
- The number of decimal places displayed for scalar values
- The increment of change in value when using stepper, slider, dial, or text style interactors
- Whether the specified increment applies to all instances of the interactor or is local to the current interactor.

Vector interactors allow you to modify these attributes for each component of the vector. Specify that component to modify by changing the Component field at the top of the vector interactor's **Set Attributes** dialog box.

You can also open the **Set Attributes** window by double clicking on the border area of the interactor inside the Control Panel.

See "Setting Interactor Attributes" on page 134 for detailed information.

Set Label
Allows you to change the title of the interactor. See "Changing the Label on the Interactor" on page 137.

Delete Deletes selected interactors.

Add Element
Brings up a cascade menu with the choices **Label** and **Separator**. These can then be added to the control panel. See "Control Panels as Dialog Boxes" on page 138.

Add Selected Interactor(s)
Adds an interactor to the Control Panel for each stand-in that is selected in the VPE window. See "Adding Interactors to an Existing Control Panel" on page 130.

Show Selected Interactor(s)
Shows the relation between a selected interactor stand-in in the VPE and the interactors in the Control Panels. See "Locating Interactor Stand-ins" on page 132.

Show Selected Tool
Shows the relation between a selected interactor stand-in in a control panel and interactor stand-ins in the VPE.

Comment...
Opens a dialog box with a text field. You can use this text field to document the use of the Control Panel. See "Adding Comments to a Control Panel" on page 133.

Control Panel Execute Menu

This execute menu is identical to that of the Visual Program Editor window. See "VPE Execute Menu" on page 160 for descriptions of the Execute options. See also 5.4, "Executing a Visual Program" on page 67.

Control Panel Panels Menu

The **Panels** menu allows you to open one or all existing control panels.

The **Panels** menu has the following options:

Open All Control Panels
Opens all existing control panels.

Open Control Panel by Name
Opens a cascade menu, from which you can select the name of the control panel to open.

Control Panel Options Menu

The **Options** menu allows you to customize the Control Panel.

The **Options** menu contains the following options:

Change Control Panel Name...
Displays a dialog box where you can enter a new name for the current Control Panel.

Control Panel Access...
Allows you to specify which control panels and control panel groups are accessible from the Control Panel's **Open All Control Panels** option under the **Windows** menu. See "Restricting Control Panel Access" on page 140.

Grid... Allows you to select the grid type you want to use on the layout area. A dialog box displays the choices. See "Customizing the VPE Window" on page 113 for more information about the **Grid...** option. See "Changing the Alignment of Interactors in the Control Panel" on page 134.

Dialog Style
Changes a control panel to "dialog style" (see "Control Panels as Dialog Boxes" on page 138).

Startup Control Panel
Allows you to specify whether the Control Panel is opened automatically when Data Explorer is started with the Image window or menu bar as the anchor window. See "Specifying a Startup Control Panel" on page 141.

Help Menu

The options in this menu are discussed in 5.3, "Using Online Help" on page 65.

Image Window Menu Bar

The Image window menu bar displays the names of six menus:

- File
- Execute
- Windows
- Connection
- Options
- Help.

Image Window File Menu

The **File** menu displays the options you use to open, save, and close visual programs, and load macros.

The **File** menu lists the following options:

Open ... Displays a dialog box for specifying a file path name to load a visual program or, if the path name ends in a directory, to show a list of visual programs in that directory. To select a program, click on the name. See "File Selection Dialog Boxes" on page 116 for more information.

Save Program
Saves the current visual program and associated Control Panels. This option saves the following files: *filename*`.net` (for network; that is, a Data Explorer visual program) and *filename*`.cfg` (for configuration). This option is grayed out unless Data Explorer was started in `-image` mode.

Save Program As...
Saves the current visual program and all associated Control Panels under a name you type in the dialog box. This option saves the following files: *filename*`.net` (for network; that is, a Data Explorer visual program) and *filename*`.cfg` (for configuration). This option is grayed out unless Data Explorer was started in `-image` mode.

Program Settings
Brings up a cascade menu with two choices:

- **Save As** saves the current settings for the visual program (e.g., window placement, interaction mode, camera viewpoint, interactor setup, etc. The settings are saved in a `.cfg` file.
- **Load** loads previously saved settings from a `.cfg` file.

Load Macro...
: Loads a macro for use, making it available in the tool palette. A file selection dialog box appears for you to specify the desired macro file name. See "Loading Macros" on page 152.

Load Module Definition(s)...
: Loads a module definition for use, making it available in the tool palette. A file selection dialog box appears for you to specify the desired module file name. The executable you are using must also include this module. See 9.2, "Loading and Using Outboard and Runtime-Loadable Modules" on page 183.

Save Image...
: Displays a dialog box that allows you to save the image and specify the format, layout, resolution, and page size of the image. See "Saving an Image" on page 94.

Print Image...
: Displays a dialog box that allows you to specify the format, layout, resolution, and page size of the print. See "Printing an Image" on page 97.

Close or **Quit**
: Closes the Image window. If you opened the Image window from the VPE, **Close** appears is the option. If you started Data Explorer with the Image window as the anchor, **Quit** is the option. **Quit** is grayed out if the **Execute** menu title is highlighted. You must select the **End Execution** option in the **Execute** pull-down menu; or if the Sequencer is running, you must stop it or pause before quitting.

Image Window Execute Menu

This execute menu is identical to that of the Visual Program Editor window. See "VPE Execute Menu" on page 160 for descriptions of the Execute options. See also 5.4, "Executing a Visual Program" on page 67.

Image Window Windows Menu

The **Windows** menu allows you to open a VPE window containing the program layout or the Control Panels associated with the visual program. All windows created from this Image window become its children. Quitting the Image window causes all of its child windows to close. Before the window closes, a dialog box appears to request confirmation.

The **Windows** menu offers the following options:

Open Visual Program Editor
: Opens a VPE window containing the current visual program. This option is disabled if you have not loaded a visual program or if the VPE is the anchor window.

Open All Control Panels
: Opens all the Control Panels and macro Control Panels for the current visual program unless the macro Control Panels are already contained in another Control Panel. This option is disabled if you have not loaded a visual program.

Open Control Panel by Name
Opens a cascade menu, from which you can select the name of the control panel to open.

Open All Colormap Editors
Opens all the Colormap Editors for the current visual program.

See 6.3, “Using the Colormap Editor” on page 119.

Open Message Window
Opens the Message Window that displays error and warning information about your execution.

See 8.2, “Using the Message Window” on page 174.

Image Window Connection Menu

The **Connection** menu allows you to monitor and control communication functions between the workstation and the server.

The **Connection** menu lists the following options:

Start Server... Connects the user workstation to the server. For more information, see 9.3, “Connecting to the Server” on page 183.

Disconnect from Server
Disconnects the user workstation from the server.

Reset Server Flushes the cache, forcing the visual program to reaccess the data on the server the next time it executes.

See “Resetting the Server” on page 185.

Execution Group Assignment...
Displays a dialog box that allows you to assign execution groups to specific machine host names.

See “Assigning Execution Groups to Workstations” on page 181.

Image Window Options Menu

The **Options** menu provides the following options, which are available only when the Image tool is used in a visual program:

View Control ...
Opens a dialog box with various view control options. For more information on this menu option, see “Controlling the Image: View Control...” on page 74.

Undo Returns to the view of the image that was displayed before the most recent action (effectively undoing the action). See “Restoring Images” on page 84.

Redo Restores the view of the image that was displayed before the last **Undo** action. See “Restoring Images” on page 84.

Reset Resets the camera to its initial position and direction. See “Restoring Images” on page 84.

AutoAxes...
Places axes around the image the next time you execute the visual program. See “AutoAxes...” on page 88.

Set Background Color...
Sets the background color of the image. See "Set Background Color..." on page 89.

Display Rotation Globe
Enables and disables the wire-framed globe display in many of the view control options. While the toggle button is pressed in, the globe display is enabled. See "Rotating the Object" on page 77.

Rendering Options...
Displays a dialog box, allowing you to choose hardware or software rendering options. See "Rendering Options..." on page 91.

Image Depth
Specifies the number of bits (8, 12 or 24) per color. See "Image Depth" on page 93.

Throttle...
Specifies the minimum number of seconds to display each frame. See "Changing the Rate of Frame Display: Throttle..." on page 93.

Change Image Name...
Opens a dialog box in that you can specify a new title for the Image window. See "Changing the Title of an Image Window" on page 93.

Control Panel Access...
Allows you to specify which control panels and control panel groups are accessible from the Image Window's **Open All Control Panels** option under the **Windows** menu. See "Control Panel Access..." on page 94.

Help Menu
The options in this menu are discussed in 5.3, "Using Online Help" on page 65.

Colormap Menu Bar
The Colormap menu bar displays the name of five menus you can use:

- File
- Edit
- Execute
- Options
- Help.

Colormap File Menu
The Colormap **File** menu provides the following options:

New	Replaces Colormap Editor values with default values.
Open...	Opens an existing Colormap as specified in the dialog box that appears. If a current Colormap is on display and changes have been made, a dialog box appears for applying wanted changes.
Save As...	Saves the current Colormap configuration as the name typed into the dialog box that appears.
Close	Closes the Colormap window.

Colormap Edit Menu

The Colormap **Edit** menu displays the options used to edit the current color map.

The following options appear on the **Edit** menu:

Undo
: Returns an operation to its previous value. **Undo** retains a stack of 10 operations.

Copy
: Copies selected control points to a buffer so that they can be pasted to another field. See "Copying and Pasting Control Points" on page 124.

Paste
: Pastes selected control points to a different field. See "Copying and Pasting Control Points" on page 124.

Use Application Default
: If the Colormap Editor is data-driven, it is still possible to override its data-driven values by explicitly setting them in the Colormap Editor. The **Use Application Default** option restores the data-driven values. It generates a cascade menu with four suboptions:

  Color (HSV) Map If a color map was provided to the color-map tab of the Colormap tool, and was subsequently modified by the user (i.e, adding, moving, or deleting control points), selection of this suboption restores the original color map.

  Opacity Map If an opacity map was provided to the opacity tab of the Colormap tool, and was subsequently modified by the user (i.e, adding, moving, or deleting control points), selection of this suboption restores the original opacity map.

  Minimum If the minimum was set either by a field passed to the data tab of the Colormap tool or by a value passed to or set for the **min** parameter of Colormap, and was subsequently modified by the user in the Colormap Editor, this suboption restores the original value.

  Maximum If the maximum was set either by a field passed to the data tab of the Colormap tool or by a value passed to or set for the **max** parameter of Colormap, and was subsequently modified by the user in the Colormap Editor, this suboption restores the original value.

  All Restores all values to their data-driven settings.

  Minimum & Maximum Restores both the minimum and the maximum values of the original color map.

Add Control Points...
: Brings up a dialog box in which you can enter the exact numerical location for control points to be added as the data value and as the value for either the Hue, Saturation, Value, or Opacity area of the color map. See "Adding Control Points" on page 122.

Constrain Horizontal
: Constrains the horizontal movement of all control points in all areas of the Colormap. See "Moving Control Points" on page 123.

Constrain Vertical
: Constrains the vertical movement of all control points in all areas of the color map.

Generate Waveforms
Displays a dialog box that allows you to select from a menu of waveforms. The waveforms can be applied to the full range of hue, saturation, value or opacity or to a selected range. See “Creating Waveforms” on page 124.

Delete Selected Control Points
Deletes selected control points in the selected area of the Colormap.

Select All Control Points
Selects all the control points in the selected area of the Colormap.

Colormap Execute Menu

This execute menu is identical to that of the Visual Program Editor window. See “VPE Execute Menu” on page 160 for descriptions of the Execute options. See also 5.4, “Executing a Visual Program” on page 67.

Colormap Options Menu

The Colormap Options menu displays the options used to edit the current color map.

The Colormap `Options` menu lists the following options:

Set Background Style to Checkerboard (or Stripes)
Changes the background bar from two vertical stripes to a checkerboard.

Note: The opacity must be less than 1 to make this pattern visible.

Axis Display
Allows you to choose between Ticks, Histogram, and Log(Histogram). See “Axis Display” on page 124.

Number of histogram bins...
Allows you to select the number of histogram bins. See “Axis Display” on page 124.

Display Control Point Data Value
Allows you to specify which control point values will be displayed. See “Display Control Point Values” on page 124

Change Colormap Name
Sets the name of the Colormap Editor. See “Changing the name of the Colormap Editor” on page 124.

Help Menu

The options in this menu are discussed in 5.3, “Using Online Help” on page 65.

Menu Bar Menu Bar

A menu bar is displayed as the anchor window when the user specifies the `-menubar` option on the command line to Data Explorer. The menu bar displays the titles of five menus:

- File
- Execute
- Connection
- Windows
- Help.

Menu Bar File Menu

The `File` menu lists the following options:

Open ... Displays a dialog box for specifying a file path name to load a visual program or, if the path name ends in a directory, to show a list of visual programs in that directory. To select a program, click on the name. See "File Selection Dialog Boxes" on page 116 for more information.

Program Settings
Brings up a cascade menu with two choices:

- `Save As` saves the current settings for the visual program (e.g., window placement, interaction mode, camera viewpoint, interactor setup, etc. The settings are saved in a .cfg file.
- `Load` loads previously saved settings from a `.cfg` file.

Load Macro...
Loads a macro for use, making it available in the tool palette. A file selection dialog box appears for you to specify the desired macro file name. See "Loading Macros" on page 152.

Load Module Definition(s)...
Loads a module definition for use, making it available in the tool palette. A file selection dialog box appears for you to specify the desired module file name. The executable you are using must also include this module. See 9.2, "Loading and Using Outboard and Runtime-Loadable Modules" on page 183.

Quit Closes the Menu Bar window

Menu Bar Execute Menu

This execute menu is identical to that of the Visual Program Editor window. See "VPE Execute Menu" on page 160 for descriptions of the Execute options. See also 5.4, "Executing a Visual Program" on page 67.

Menu Bar Connection Menu

The `Connection` menu allows you to monitor and control communication functions between the workstation and the server.

The `Connection` menu lists the following options:

Start Server... Connects the user workstation to the server. For more information, see 9.3, "Connecting to the Server" on page 183.

Disconnect from Server
Disconnects the user workstation from the server.

Reset Server Flushes the cache, forcing the visual program to reaccess the data on the server the next time it executes.

See "Resetting the Server" on page 185.

Execution Group Assignment...
Displays a dialog box that allows you to assign execution groups to specific machine host names.

See "Assigning Execution Groups to Workstations" on page 181.

Menu Bar Windows Menu

The `Windows` menu allows you to open a VPE window containing the program layout or the Control Panels associated with the visual program. All windows created from this Image window become its children. Quitting the Image window causes all of its child windows to close. Before the window closes, a dialog box appears to request confirmation.

The `Windows` menu offers the following options:

Open Visual Program Editor
Opens a VPE window containing the current visual program. This option is disabled if you have not loaded a visual program or if the VPE is the anchor window.

Open All Control Panels
Opens all the Control Panels and macro Control Panels for the current visual program unless the macro Control Panels are already contained in another Control Panel. This option is disabled if you have not loaded a visual program.

Open Control Panel by Name
Opens a cascade menu, from which you can select the name of the control panel to open.

Open All Colormap Editors
Opens all the Colormap Editors for the current visual program.

See 6.3, "Using the Colormap Editor" on page 119.

Open Message Window
Opens the Message Window that displays error and warning information about your execution.

See 8.2, "Using the Message Window" on page 174.

Help Menu

The options in this menu are discussed in 5.3, "Using Online Help" on page 65.

Message Window Menu Bar

The Message window menu bar displays the names of five menus:

- File
- Edit
- Execute
- Options
- Help

Message Window File Menu

The Message window `File` menu provides the following options:

Clear	Clears the Message window.
Log...	Allows you to designate a file that will receive all subsequent messages displayed in the Message window.
Save As...	Saves the current contents of the Message window to the file named in the dialog box that appears.
Close	Closes the Message window.

Message Window Edit Menu

The Message window `Edit` menu displays the options used to edit the current Message window.

The following options appear on the `Edit` menu:

Next Error
: Displays and highlights the next error in the Message window and highlights the tool that reported it.

Previous Error
: Displays and highlights the previous error in the Message window and highlights the tool that reported it.

Message Window Execute Menu

This execute menu is identical to that of the Visual Program Editor window. See "VPE Execute Menu" on page 160 for descriptions of the Execute options. See also 5.4, "Executing a Visual Program" on page 67.

Message Window Commands Menu

The Message window `Commands` menu displays the options for running specific commands. The following options appear on the `Commands` menu:

Debug Tracing
: When enabled, displays instance numbers on each tool in a visual program (to distinguish between multiple instances of a tool) and turns on tracing in the message window so that as each module is run its name and instance number is displayed.

Execute Script Command
: Allows commands to be issued directly to the server (see "Executive" on page 126 in *IBM Visualization Data Explorer User's Reference*).

Show Memory Use
: Displays the current memory usage (see "Usage" on page 362 in *IBM Visualization Data Explorer User's Reference*).

Message Window Options Menu

The Message window `Options` menu displays the options used to edit the current Message window.

The Message Window `Options` menu lists the following options: See also 8.2, "Using the Message Window" on page 174.

Information Messages
: Allows you to either display or not display Information messages.

Warning Messages
: Allows you to either display or not display Warning messages.

Error Messages
: Allows you to either display or not display Error messages.

8.2 Using the Message Window

The Message window allows you to monitor the progress of a visual program as it executes. It displays information, warning and error messages from the executive and output from the Print and Echo modules (if you include them in your visual program).

You can use the Print module, for example, to print the contents of an object. See Print in *IBM Visualization Data Explorer User's Reference* for more information. You can use the Echo module to print values or strings. See Echo in *IBM Visualization Data Explorer User's Reference*.

To open the Message window, select the **Open Message Window** option from the **Windows** pull-down menu of either the Image window or the VPE window. The Message window consists of a text window and various pull-down menus. When a visual program is running, any information, warnings, or errors sent by the server to the user interface are shown in the display area. (See also "Error Messages" on page 71.)

The Message window allows users to save the contents of the text window in two different ways. The **Save As...** command of the Message window's **File** menu brings up a File Selection dialog box that allows the user to designate a file to which the current contents of text window are to be saved. Alternatively, the **Log...** command of the **File** menu brings up a File Selection dialog box that allows the user to designate a file that will receive all subsequent messages displayed in the text window.

The Message window provides a number of techniques to help the user locate the source of errors and warnings. Before each successful execution of the visual program, the string "Begin execution" is placed in the text window. This allows the user to more easily determine whether messages are the result of the most recent execution. Also, any line in the text window can be highlighted before execution. Highlighted lines remain highlighted over executions, making it easy to locate a marker indicating the beginning of execution. This technique is particularly useful when a large amount of information (from Print or Echo) is being displayed in the Message window. Alternatively, the contents of the text window can be removed with the **Clear** option of the Message window's **File** menu.

When an error occurs during the execution of a module, the errant module can be located in the VPE by double-clicking on the line in the Message window's text window containing the error message. This selection causes the VPE's canvas to shift so that the indicated module is within the canvas's scrolled viewing area. Errors for the last execution can also be located using the **Next Error** and **Previous Error** commands of the Message window's **Edit** menu. These commands expose and highlight errors that occurred before or after the currently highlighted error. If no error is currently highlighted, then **Previous Error** indicates the last error during execution and **Next Error** indicates the first error.

The Message window can be configured so that it does not display certain message types in the text window (by using the **Error Messages**, **Warning Messages**, and **Information Messages** buttons in the Message window's **Options** pull-down menu). If a toggle button is activated (colored), the corresponding message type is displayed. The default behavior is to display all error, warning, and information messages. You can change this default behavior with the `infoEnabled`, `warningEnabled`, and `errorEnabled` configuration options described in Appendix D, “User Interface Configuration” on page 299. By default, the Message window pops up when an error is displayed in the text window and does not automatically pop up when information or warning messages are displayed. You can also change this default behavior with the `infoOpensMessage`, `warningOpensMessage`, and `errorOpensMessage` configuration options described in Appendix D, “User Interface Configuration” on page 299.

Lastly, when connected to the server, the **`Execute Script Command...`** command of the Message window’s **`Options`** menu allows commands to be issued directly to the server. However, only advanced users should use this feature, as the results of the commands entered can upset the state of the visual program. This command brings up the **`Execute Script Command`** dialog box, which will accept a single command in a 1-line text window. Any messages that result from this command appear in the Message window’s text window.

Chapter 9. Graphical User Interface: For Advanced Users

9.1 Using Distributed Computation

Data Explorer provides you with the capability to distribute your visual program across a network of heterogeneous workstations. Distributing your visual program provides you with parallelism and enhanced resource utilization. Parallelism is achieved by the simultaneous execution of different portions of the visual program on each of the workstations. The amount of parallelism that you can achieve is dependent on the organization of your visual program and the number of available workstations.

Enhanced resource utilization can be achieved, for example, by assigning computationally intensive portions of the visual program to the more powerful workstations. If the data you are visualizing is located on one or more workstations, then performing some of the data realization and transformation on the workstations containing the data can reduce data transfer overheads.

Distributed processing in Data Explorer is achieved in two ways: by using "outboard" modules (user supplied) or by placing groups of tools to "execution groups". These two methods can be used independently or in combination. For a discussion of outboard modules see 10.3, "Inboard, Outboard, and Runtime-loadable Modules" on page 85 in *IBM Visualization Data Explorer Programmer's Reference*. Execution groups can be created and modified using the Visual Program Editor or by using attributes if you are using the Data Explorer scripting language (see Chapter 10, "Data Explorer Scripting Language" on page 187). Once the execution groups are created, you assign each group to the workstations over which you wish to distribute the visual program. You can assign more than one group per workstation. Note that if more than one group is assigned to a given workstation, the groups will not be run as separate processes.

Data Explorer uses this two part approach of creating groups and assigning groups to make it easier for you to change the set of workstations over which you distribute your visual programs. This utility is especially convenient if you share visual programs with other users.

When you execute a visual program for the first time, the Data Explorer executive is started on each workstation over which the program is to be distributed. Each executive "plans" the execution and executes each of the execution groups assigned to it. This means that not only is the computation and data flow distributed, but the control flow is distributed as well. One of the workstations is a "master"—the workstation to which the user interface is connected. The master creates and initiates the communication between the other workstations and distributes commands from the user interface to all the workstations. The master also executes any execution group that is not assigned to another workstation in addition to its own assigned groups.

Creating, Modifying, and Deleting Execution Groups

By default, all tools in a visual program belong to a single unnamed execution group. This group is executed on the master workstation. New execution groups are created and existing execution groups are modified and deleted using the **Execution Group** dialog box (Figure 79 on page 179).

The dialog box consist of three parts:

Groups

Displays the current set of execution groups. A group name can be selected by clicking on the name. It can be deselected by clicking on the name a second time or by clicking on another name.

Name

A text field for specifying the name of a new execution group, the name of an execution group to modify or the name of an execution group to be displayed. If an execution group is selected, the name of the group is displayed in the **Group Name** text field.

Pushbuttons

Create causes a new execution group to be created. **Add To** causes a set of selected tools to be added to an existing execution group. **Remove From** causes a set of selected tools to be removed from an execution group. **Delete** causes an existing execution group to be deleted (the tools in the group are not deleted from the VPE canvas). **Show** causes all the tools on the canvas that are members of the selected group to be selected. **Close** causes the dialog box to be closed.

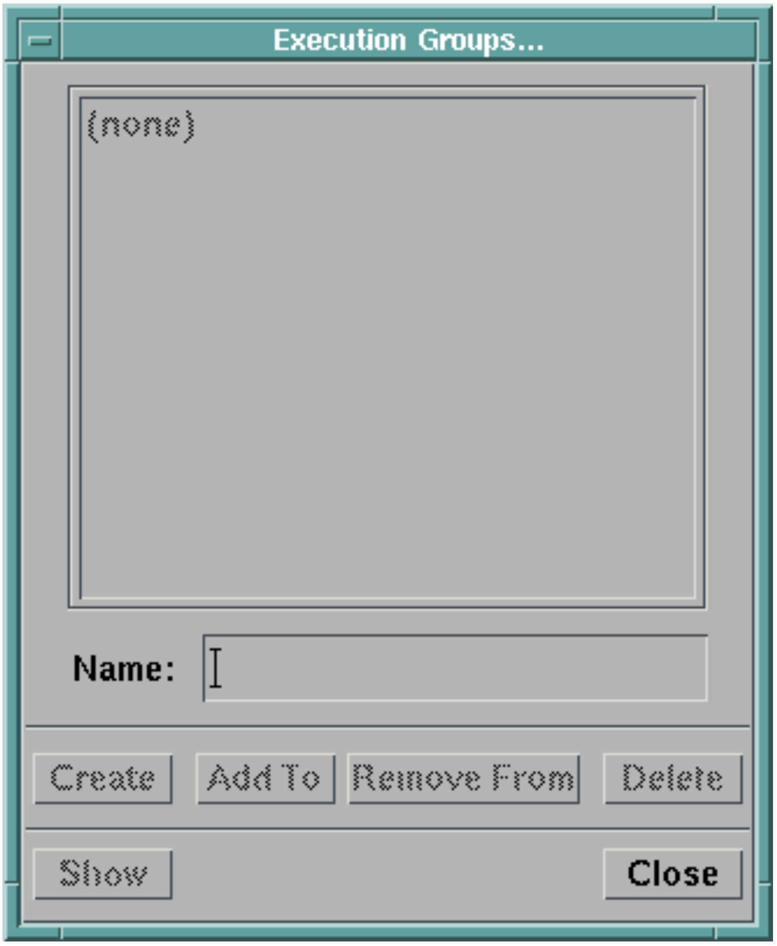


Figure 79. Execution Group Dialog Box

Creating an Execution Group

To create an execution group

1. Select the tools on the VPE canvas that are to be placed in a single execution group. This can be done by a rubber-band selection or using a shift-click selection or a combination of both.
2. Open the **Execution Group...** dialog box by selecting the **Edit** menu **Execution Group...** option.
3. In the **Group Name** text field, enter a name for the execution group.
4. Click on the **Create** button.

The newly created execution group will appear in the list of execution groups.

Note: If a tool is a member of an existing execution group and it is placed into a new group, the tool is automatically deleted from the existing group. A single tool can not be a member of two execution groups.

Modifying Execution Groups

Execution groups can be modified by adding new tools to the group or removing tools from the group.

To add tools to an Execution Group:

1. Select the tool or tools to be added to a group.
2. Open the **Execution Groups...** dialog box by selecting the **Edit** menu **Execution Groups...** option.
3. Select the name of the execution group to which the tool(s) are to be added.
4. Click on the **Add To** button.

To remove tools from an Execution Group:

1. Select the tool or tools to be removed from a group.
2. Open the **Execution Groups...** dialog box by selecting the **Edit** menu **Execution Groups...** option.
3. Select the name of the execution group from which the tool(s) are to be removed.
4. Click on the **Remove From** button.

Deleting an Execution Group

To delete an execution group:

1. Open the **Execution Groups...** dialog box by selecting the **Edit** menu **Execution Groups...** option.
2. Select the name of the execution group to be deleted.
3. Click on the **Delete** button.

Displaying the Tools in an Execution Group

Data Explorer provides you with the ability to display all the tools that are members of an execution group. To display the tools:

1. Open the **Execution Groups...** dialog box by selecting the **Edit** menu **Execution Groups...** option.
2. Select the name of the execution group to be displayed.
3. Click on the **Show** button.

The tools that are members of the execution group become selected. If the selected tools are not part of the currently displayed portion of the visual program, the VPE will be updated so that the selected tools will be displayed.

Assigning Execution Groups to Workstations

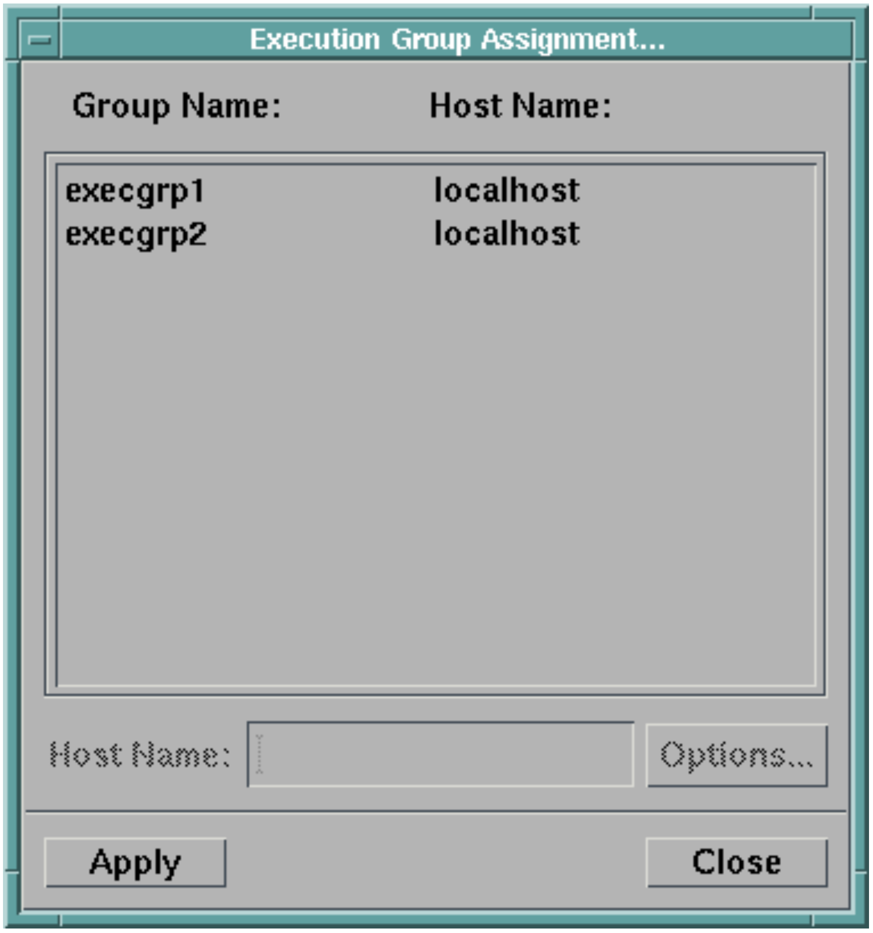
Once you have decomposed your visual program into execution groups, you can assign these groups to workstations (or hosts). If you do not specify a host for a particular execution group, the group will be executed on the master. Execution groups are assigned to a host using the **Execution Group Assignment...** dialog box (Figure 80 on page 182).

The dialog box consists of three parts:

Groups and Hosts
Displays the name of an execution group and the host on which the group is executed. If a host name is not given, then “localhost” is displayed as the host name.

Host Name A text field for specifying the name of a host on which an execution group is to be executed.

Pushbuttons
OK causes the assignments of executions groups to host names and closes the dialog box. **Options...** opens a dialog box that allows you to specify startup options (e.g., memory size, or the Data Explorer executive that will be started on the host.



execgrp1 localhost
execgrp2 localhost

Figure 80. Execution Group Assignment Dialog Box

To assign an execution group to a workstation:

1. Open the **Execution Group Assignment...** dialog box by selecting the **Connection** menu **Execution Group Assignment...** option in either the Visual Program Editor or Image Window.
2. Select the execution group that you wish to assign to a host by clicking on its name.
3. Enter the host name in the **Host Name** text field and hit Enter.
4. Click on the **Options...** button to specify any options for the host, for example memory size.
5. If there are more groups to assign to hosts repeat Steps 2 through 4, otherwise, click on **OK**.

Restrictions

In general, you can place any tool in any execution group, alter the members of an execution group at any time, or change the assignment of an execution group from one host to another at any time. The exceptions to these rules deal with:

- Modules maintaining state (e.g., Streakline)
- Hardware rendering
- The Pick Module.

Modules that maintain internal state information using private cache objects cannot be freely moved between hosts. Doing so will cause the module to lose its state information or obtain old state information. If you do reassign a module maintaining state to another host, you should reset the server using the **Connection** menu **Reset Server** option to ensure correct execution. The only Data Explorer module provided with the system that maintains state information is Streakline.

When you are using the hardware rendering option with either the Display or Image tools, the tool should be assigned to execution on the host physically connected to your display.

The Pick module must be run on the same host as the Image tool.

9.2 Loading and Using Outboard and Runtime-Loadable Modules

Data Explorer allows the following types of user-written module:

inboard modules are compiled and linked into the Data Explorer Executive.

outboard modules are run as separate processes.

runtime-loadable modules are loaded at runtime and effectively become inboard modules.

For an outboard or runtime-loadable to be available for use, its module description file must be loaded into the tool palette. To load a module description file:

1. Select the **Load Modules Description(s)...** option from the **File** pull-down menu in the VPE. A file selection dialog box appears. For information about how to use the dialog box, see “File Selection Dialog Boxes” on page 116.
2. Use the dialog box to locate the description file for the module you want to load, and select the desired file. The module name appears on the tool palette under its assigned category.

   Note: If the module description you loaded has the same name as an module description loaded previously, the more recent description replaces the less recent, regardless of whether they were assigned to the same category.

When the new module is available for use, it is listed in the tool palette as a member of the category you assigned it to.

You can configure Data Explorer to automatically load module descriptions when you begin the Data Explorer program. To do this, use the `-mdf` *filename* option when starting Data Explorer, where *filename* specifies one or more module definitions. Alternatively, you can use the *DXMDF* environment variable. For more information on how to do this, see “Using Environment Variables” on page 59 and C.1, “Environment Variables” on page 292.

9.3 Connecting to the Server

Before you can execute a visual program, your workstation must have established connection to the server. When you start Data Explorer, the program automatically starts the server connection, unless you specified that only the user interface was to be started. However, if for some reason your workstation becomes disconnected between the time you start Data Explorer and the time you are ready to run a program, you must reestablish connection with the server.

To start the server connection, select the **Start Server...** option from the **Connection** pull-down menus in the VPE or Image window. A dialog box appears (Figure 81 on page 184).

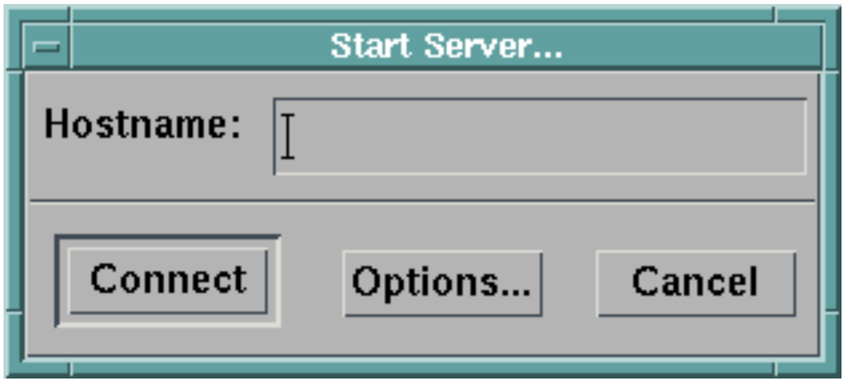
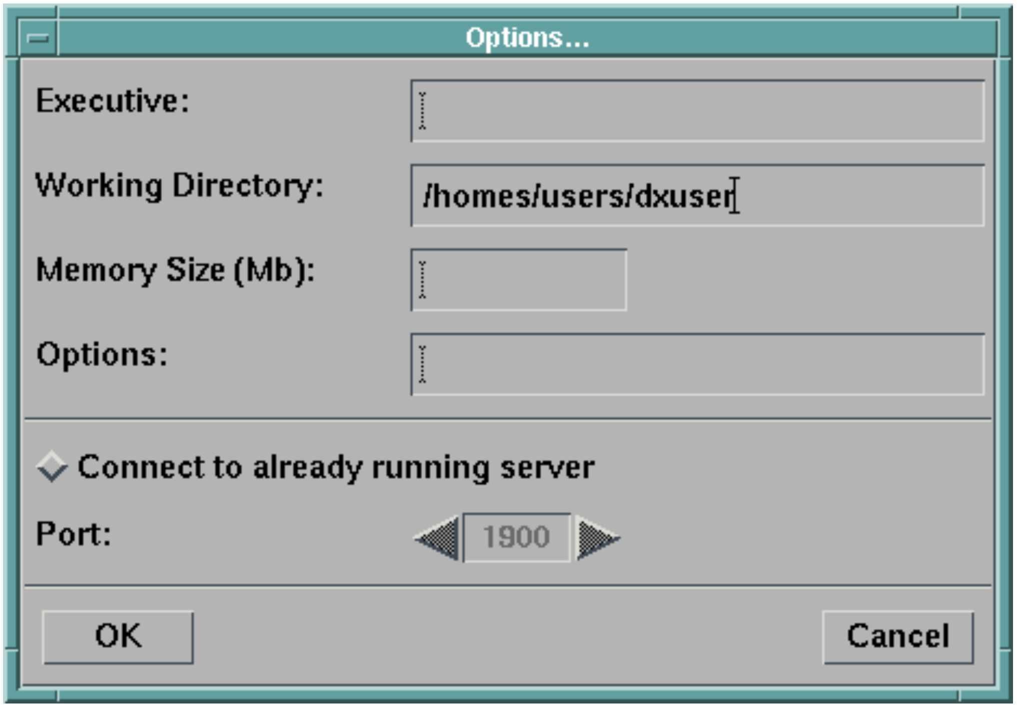


Figure 81. Start Server Dialog Box

The Hostname field, by default, contains the name "localhost." If the DXHOST environment variable or the **-host** argument was specified, this field contains the setting of that variable.

You can change the Hostname field by clicking on it and typing in the new name. When it displays the desired name, connect to the server by clicking on the **Connect** button.

To change the options associated with starting the server, click on the Options... button. This opens the **Options...** dialog box (Figure 82). Most of the time, it is not necessary to use these options.



Working Directory: /homes/users/dxuser

Port: 1900

Figure 82. Start Server Options Dialog Box

The fields of this dialog box are as follows:

Executive
: Specifies the name of the executive to run. You might change this field if you want to run a customized version of Data Explorer.

Working Directory
: Specifies the default directory to search for files.

Memory Size
: Specifies the amount of memory to use, in megabytes.

Options
: Specify options in this text box as you would on the command line when invoking Data Explorer; that is, each option must be preceded by a dash.

  -cache on saves the state of the visual program in memory after an execution. If the visual program changes, Data Explorer reexecutes only the portion of the program affected by the change. If **-cache off** is specified, Data Explorer must reexecute the entire program for each change. (The default setting is on.)

  -trace on displays each execution step, as it happens, in the **Debug** window. It is mainly used for debugging purposes. (The default setting is off.)

  -log on saves all communication between the server and the user interface to a file. (The default setting is off.)

  Other options are listed in C.2, "Command Line Options" on page 295.

Connect to already running server radio button
: Specifies that the user interface should connect with a server that was started earlier.

  You might use this option to set up a connection to a debugging session for a customized version of the Data Explorer executable. You can specify the port number in using the stepper below the button. Note that when this button is toggled on, all other fields in this dialog box become disabled.

Resetting the Server

If the data set your visual program is using changes (e.g., by being edited) during your Data Explorer session, and the **cache** is enabled (the default condition), it may be necessary to force Data Explorer to reinitialize the server executive to access the new data. To do this, select the **Reset Server** option from the **Connections** pull-down menu.

The action of resetting the server flushes the executive cache. The next time you execute the visual program, it executes the entire network, not just the portions affected by changes internal to the Data Explorer session, and will thus reaccess the data set.

Chapter 10. Data Explorer Scripting Language

When you create a visual program with the graphical user interface, Data Explorer saves the program in a `.net` file. This saved version is actually a set of scripting-language commands. You do not need to understand the script language unless you are running Data Explorer in script mode.

In Data Explorer script mode you can also perform tasks that would be awkward with a visual program (e.g., facilitation of batch processing or debugging of a module).

10.1 Starting Data Explorer in Script Mode

To run Data Explorer in script mode on a workstation, you must have an account on that workstation.

To start Data Explorer, follow these steps:

1. Start the X Window System session on the workstation.
2. Type:

   ```
   dx -script
   ```

   When script mode starts, you will see a prompt symbol (`dx>`), indicating that Data Explorer is ready to accept input. (If you want to change the prompt symbol, see 10.7, "Using Data Explorer Script Commands" on page 206.)

(All of the command line options for Data Explorer are described in C.2, "Command Line Options" on page 295.)

You can type commands directly at the command line, but you may find it more convenient to create a script and submit it to Data Explorer for execution. To submit a script, type:

```
include "scriptname"
```

at the prompt, where *scriptname* is the name of the script.

Once you have submitted a script, Data Explorer will process the commands it contains. Note that none of the direct interactor options are available in script mode: you must use the Image tool in the graphical user interface to take advantage of those options.

After the included script has been processed, you can include another script. To terminate your Data Explorer session, type:

```
quit
```

You can also include a script name directly in the script command: add the name of the script after the **`-script`** option. Data Explorer will terminate automatically when it has executed the script. For example, type:

```
include "/usr/lpp/dx/samples/scripts/scriptexample"
```

You will see a sequence of images created with sequencer commands. The directory **`/usr/lpp/dx/samples/scripts`** contains examples for many modules. You may find it helpful to experiment with them to learn how they function.

Note: To ensure that an example program does not exit before you want it to, invoke script mode first and then "include" the program. Otherwise, some programs will execute and disappear so quickly that you won't be able to identify the image.

Setting Environment Variables

There are several environment variables that you may find useful to customize Data Explorer. These can be set in your login profile.

DXDATA: The DXDATA environment variable specifies a list of directories in which Data Explorer will search for data files. If the data you wish to import is in one of the directories specified in the DXDATA environment variable, you do not need to provide the complete path name to the Import tool. Specify the file name, and the Import module will look in the specified directories for the data file. The directories will be searched in the order in which they are listed in the environment variable; and the first occurrence of the data file will be used.

An example of a statement that sets the DXDATA environment variable (in the C shell environment) is the following:

```
setenv DXDATA /usr/mydirectory/mydata:/usr/group/groupdata
```

where **/usr/mydirectory/mydata** and **/usr/group/groupdata** are two directories that contain data files. Multiple directories can be listed, with each directory name separated by a colon.

DXHOST: The DXHOST environment variable is the initial machine name of the server on which to run the executive. If DXHOST is not specified, then a default of “localhost” is used. See 9.3, “Connecting to the Server” on page 183 for more information on how to connect to the server. The host name should be the name that results when you issue the **uname -n** shell command.

DXINCLUDE: If this environment variable is set, Data Explorer looks for included scripts first in the current directory, and then in each of the directories specified in the colon-separated list specified by this variable.

DXMACROS: The DXMACROS environment variable is a list of the directories in which Data Explorer will look for macros.

10.2 Understanding the Script Structure

The following example illustrate some of the more important characteristics of scripts; a detailed description of each of the elements follows. However, you may prefer to simply study these examples (and perhaps those in /usr/lpp/dx/samples/scripts) and then begin writing your own scripts.

Example 1. A Simple Script

In this example, the data found in /usr/...cloudwater is imported and assigned to the variable **data**. Then the Isosurface module is called on **data** (with no other parameters set) and the result is assigned to **iso**. A Camera is created using AutoCamera, and the isosurface is displayed using Display (note that the Image tool is not available in the scripting language.

```
data = Import("/usr/lpp/dx/samples/data/cloudwater");
iso = Isosurface(data);
camera = AutoCamera(iso);
Display(iso, camera);
```

Example 2. Setting Parameters

Suppose that in the previous example we wished to set the Isosurface “number” to 3. **number** is the third parameter to Isosurface. We can replace the second line of the script in Example 1 with:

```
iso = Isosurface(data, NULL, 3);
```

or, alternatively,

```
iso = Isosurface(data, number=3);
```

Example 3. Using a Macro

It is possible to create and use macros in the scripting language. A macro is defined using the keyword “macro,” as in the following example.

```
macro make_iso(data, isovalue) -> (isosurface)
{
   isosurface = Isosurface(data, isovalue);
}
```

To use the macro, simply call it with the required parameters:

```
iso1 = make_iso(data, 0.1);
iso2 = make_iso(data, 0.2);
...
```

A macro can have as many inputs or outputs as desired. Note that it is not necessary to pass parameters into a macro; the parameters will be found in the environment outside of the macro if necessary. However, it *is* necessary to pass any parameters *out* of the macro that are intended to be used outside of the macro.

Example 4. Using Route in the Script Language

The Route module is used to choose between different destinations for a particular object. For example, you could choose to either write an image to a file or display the image to the screen.

In order to use Route in a script, the Route module and the tools that consume the outputs of Route must be contained in a macro.

```
data = Import("/usr/lpp/dx/samples/data/cloudwater");
iso = Isosurface(data);
camera = Autocamera(iso);
image= Render(iso, camera);

macro do_which(which, image)
{
   image_to_display, image_to_write = Route(which, image);
   Display(image_to_display);
   WriteImage(image_to_write);
}
do_which(1, image);
```

The call to the macro do_which with a value of 1 causes the first output branch (Display) to be executed. WriteImage is *not* executed. If do_which had been called with a value of 2, however, then WriteImage (and not Display) would have been executed.

Example 5. Using the Sequencer

You can use the Sequencer in script mode. The special variables you use are:

@startframe the starting integer of the sequence

@endframe the ending integer of the sequence

@deltaframe the increment between frames (default = 1)

@frame contains the sequence number of the current frame.

The keyword "sequence" identifies the macro that will be run each time @frame is incremented, and the keyword "play" will start the sequence.

The following script will call the macro "doit" with the values 0, 2, 4, 6, 8, 10:

```
@startframe =0;
@endframe =10;
@deltaframe =2;
macro doit(i)
{
   Echo(i);
}
sequence doit(@frame);
play;
```

10.3 Language Delimiters

As the preceding examples show, the Data Explorer scripting language resembles a conventional programming language. Unlike some programming languages that treat all characters as uppercase, the scripting language is case sensitive. Also, you can type statements beginning at any column in the line. This allows you to indent sections to clarify the program structure.

Data Explorer uses the following characters to separate or delimit elements of the scripting language:

; A semicolon terminates a script statement.

, A comma separates keywords, arguments, lists, or vectors.

[ ] Brackets enclose vectors, matrices, and tensors.

{ } Braces enclose lists and blocks of statements in macros.

In some places, one or more blank spaces can be used in place of a comma (e.g., in separating elements of a vector).

Commenting Scripts

A *comment* is defined as two slashes (//) followed by a sequence of characters and terminated by the end of the line. Comments have no effect on the script other than to enhance its readability. For example:

```
// This is a valid comment
```

Naming Variables and Macros

You can name the variables and macros with *identifiers*. Identifiers are sequences of characters selected from the following:

- Uppercase alphabetic characters (**A-Z**)
- Lowercase alphabetic characters (**a-z**)
- Numerals (**0-9**)
- Special characters:
  - Underscore (_)
  - Single quote (**'**)
  - "At" sign (**@**)

All identifiers must start with either an alphabetic character, an underscore (_), or an "at" sign (@). Remember that the Data Explorer script language is case sensitive: identifiers that differ in the case of at least one character are considered to be different identifiers. Identifiers are currently limited to a length of 200 characters.

Note: IBM reserves the definition and use of most identifiers that begin with @. However, there are some built-in @ variables that you can set; these are discussed in 10.7, "Using Data Explorer Script Commands" on page 206.

The following are valid, unique identifiers:

```
ComputeSine
Compute_Sine
c0mpute_sine
compute_s1ne
compute_sine'
```

Identifiers can be used as:

- variable names
- function names.

Some identifiers are reserved and cannot be used as a variable or a function name. Every other identifier can be used both as a variable and as a function name. The proper use of the identifier is determined by its context.

Reserved Words

The identifiers in the following list cannot be used for variable or function names. Those marked with asterisks are reserved for use in future releases.

and*	**loop**	**repeat***
backward	**macro**	**or***
cancel*	**not***	**sequence**
else*	**off**	**step**
false*	**on**	**stop**
for*	**palindrome**	**then***
forward	**pause**	**true***
if*	**play**	**until***
include	**quit**	**while***

The identifier **NULL** is a reserved variable name. It can be used to initialize other variables or function arguments to have no value assigned to them.

Specifying Values in a Script

You can specify values in a Data Explorer script as any of the following:

- String constants
- Scalar numeric constants
- Vectors, matrices, and tensors
- Lists

You can use any of the formats described in the following sections to specify values to Data Explorer. To have Data Explorer use a particular format to echo these values to you, you must first format a string with the Format module (as described in “Format” on page 146), and then echo the string using the Echo module.

String Constants

String constants consist of a sequence of any characters delimited by the double quote character ("). However, a null character in a string delimited by double quotation marks (for example, “str\0ing”) causes the string to be terminated at the null character.

String constants have the following characteristics:

- They are delimited with double quotes.
- They can be up to 4000 characters long.
- They may extend over multiple lines, providing that the last character on each line (except for the last line) is a backslash. For example, the following lines,

  ```
  a = "123\
  456";
  ```

  are equivalent to

  ```
  a = "123456";
  ```

You can use the following escape sequences to include special characters in a string constant:

Description	Character	Escape Sequence
newline	NL (LF)	**\n**
horizontal tab	HT	**\t**
vertical tab	VT	**\v**
backspace	BS	**\b**
carriage return	CR	**\r**
formfeed	FF	**\f**
audible alert	BEL	**\a**
backslash	\	\\
question mark	?	**\?**
single quote	'	**\\'**
double quote	"	**\\"**
octal number (*ooo*)		**\***ooo*
hex number (*hh*)		**\x***hh*

The following are examples of valid string constants:

```
""          // an empty string

"this a string:  ~!@#$%^&*()_+"
```

Scalar Numeric Constants

Scalar numeric constants are sequences of numeric characters that can be used in two ways:

- As values themselves
- As components of a vector, matrix, tensor, or list.

There are two kinds of scalar numeric constants:

- Integer
- Floating point

The following sections describe these constants.

Integer: *Integers* are the set of counting numbers, or their negatives (e.g., 0, 1, 2,...). By virtue of their 32-bit internal representation, integer values range from -2^{31} to $2^{31} - 1$. They can be prefixed with a minus sign (–) to represent a negative number. Integers in Data Explorer can be represented in the following base systems:

Decimal
Decimal notation (base 10) is the most common notation for integers. Decimal numbers are constructed from sequences of numerals (0, 1, ..., 9).

Octal
When a sequence of numerals begins with the numeral zero (0) followed by a numeral from 0 to 7, Data Explorer treats it as an *octal*, or base-8, number. If a numeric sequence starts with a zero but contains either an 8 or 9, then these digits are identified as invalid octal digits. They are, however, correctly converted. For example, although the following octal numbers are both converted to the decimal number 17, the first produces an error message, but the second does not:

```
019
021
```

Hexadecimal
Hexadecimal, or base 16, numbers can be constructed from both the numerals (0 to 9) and the extended hex-digits (a to f, or A to F). To differentiate them from decimal and octal integers, hexadecimal numbers start with either the sequence **0x** or the sequence **0X** (the numeral zero followed by the letter X).

The following are examples of valid integers, all of which have the value 95 base 10:

```
  95
0137
0x5f
```

Floating Point: *Floating-point* numbers are used to represent the set of real numbers. These numbers encompass both rational and irrational numbers. By virtue of their 32-bit, IEEE single-precision internal representation, they lie in the range of $\pm 3.4028 \times 10^{38}$. The smallest step between values is $\pm 1.1754 \times 10^{-38}$. Like integers, floating-point numbers can be prefixed by a minus sign (**–**) to represent a negative number. Floating-point numbers can be expressed in two ways:

Standard representation

The *standard representation* of a floating-point number consists of a decimal number followed by a decimal point (**.**), followed by another decimal number. The first decimal number represents the whole part of the floating-point number. The second represents the fractional part. Either the first or the second of the numbers surrounding the decimal point can be omitted, but not both. If the first is omitted, then the number is purely fractional. If the second is omitted, then the number does not contain a fractional part. This second alternative is useful for representing integer values that lie outside of the range representable by the integer format.

Scientific notation

Scientific notation is an alternative means of representing floating-point numbers. A number in scientific notation has the form *x*e*y* (or *x*E*y*). The number *x* can be either a standard floating-point number or a decimal integer. The number *y* must be a decimal integer. It can be prefixed by a minus sign. This scientific notation is simply shorthand for writing $x \times 10^y$. The effect of the decimal value *y* is to specify the number of places the decimal point should be shifted to the right, or if *y* is negative, to the left.

The following are examples of valid floating-point numbers, all of which have the value 95.0:

```
95.
95.0
95e0
9.5E1
950e−1
9.50e+1
```

Vectors, Matrices, and Tensors

Vectors, *matrices*, and *tensors* are higher dimensional mathematical entities that are used for the representation of specific kinds of data.

Vectors: A vector is a quantity that has both magnitude and direction in *n*-dimensional space. It corresponds to a directed line segment whose length represents the magnitude of the vector and whose orientation corresponds to its direction.

Vectors are composed of a sequence of scalar values enclosed by square brackets **[ ]**. The scalar values can be separated by commas if desired, although this is not necessary. If the elements of a vector are not homogeneous, (e.g., if they are both integer and floating-point elements), then the integer elements are converted to floating point. The following are all valid vectors:

```
[0.0 0.0 0.0]              // the origin of a 3-D space
[0, 0, 1]                  // an axis in a 3-D coordinate system
[1, 1.0  1, 1.0  1]        // a vector in a 5-D space
```

Matrices: Matrices are 2-dimensional collections of scalars. They are used to represent, among other things, the coefficients of a set of simultaneous equations or a transformation of a vector.

Matrices are constructed from a sequence of vectors enclosed by square brackets. Each of the vectors contained in a matrix must have the same length as all of the others. The following are all valid matrices:

```
[[1 0 0 0][0 1 0 0][0 0 1 0][0 0 0 1]]          // a 4x4 identity matrix

[[ 0.707 0.707 0.000]          // a 45-degree rotation
 [–0.707 0.707 0.000]          //     about the Z axis
 [ 0.000 0.000 1.000]]

[[1 1 1 1][2 2 2 2]]           // a 2x4 matrix
```

Tensors: Tensors are a generalization of the concept of vectors. On one hand, the elements in a tensor have meanings that are independent of the coordinate system in which they are embedded. On the other hand, one can associate certain metrics to them that vary among coordinate systems.

In general, a rank *n* tensor can be formed by surrounding *k* rank *n*–1 tensors with square brackets. (Note that scalars, vectors, and matrices are rank 0, 1, and 2 tensors, respectively.) As with the matrices, all of the subtensors must have the same shape.

The following are valid tensors:

```
[[[[[0xabcd]]]]]           // a 1x1x1x1x1 rank 5 tensor

[[[1 0 0]                 // a 3x3x3 rank 3 tensor with
  [0 0 0]                 //        1's on the diagonal
  [0 0 0]]
 [[0 0 0]
  [0 1 0]
  [0 0 0]]
 [[0 0 0]
  [0 0 0]
  [0 0 1]]]
```

Lists

Unlike the vector, matrix, and tensor constructions that aggregate several lower dimensional data elements into a single higher one; the *list* construction collects several homogeneous elements together so that they can be handled as a single entity while still retaining their individuality.

Lists are constructed by enclosing a sequence of scalars, vectors, matrices, rank *n* tensors, or string constants in braces (**{ }**). The elements of a list can be separated by commas, although they need not be. In Data Explorer, a list is the same as an Array (see “Arrays” on page 28).

The following are examples of valid lists:

```
{1.0 2.0 3.0}            // 3 scalar values (for isovalues)

{[0.0 0.0 0.0],          // 4 vector values for use as
 [1.0 0.0 0.0],          //     streamline seed points
 [2.0 0.0 0.0],
 [3.0 0.0 0.0]}

{“a” “list” “of” “string”
“constants”}
```

Lists of scalars can also be defined with a convenient shorthand notation that specifies the following:

- The list’s starting value
- The list’s ending value
- A *stepping increment* (optional).

If you do not specify a stepping increment, then the default is 1. If any of the values in the list constructor (including the stepping increment) are specified as floating point numbers, then the generated list contains floating-point numbers; otherwise, it contains integers. If the starting value is smaller than the ending value, the list elements are generated in increasing order; otherwise they are generated in decreasing order. Also, only the magnitude of the stepping increment is important, not the sign. A negative stepping increment produces the same results as a positive one.

The values included in the list are generated by continually adding the value of the stepping increment to the starting value until the resultant value passes the ending value. Each of the following produces the same list:

```
{-1 1 3 5 7 9}
{-1 .. 9 : 2}
{-1 .. 9 : -2}
{-1 .. 10 : 2}
```

Note: Spaces are required around the .. operator.

Lists specified using this notation will be represented as a *Regular Array* of 1-vectors. See “Arrays” on page 28 for a discussion of Array types.

10.4 Building Expressions and Statements

You can use the basic elements of the Data Explorer scripting language to build expressions and assignment statements. Most statements in the scripting language are assignment statements; however, a special group of script commands with their options can form a statement. These commands are described in 10.7, “Using Data Explorer Script Commands” on page 206. The following sections tell you how to write Data Explorer script expressions and statements.

Arithmetic Expressions

You can combine scalar values and variables that contain values with the arithmetic operators listed below to derive new values. Lists of scalars can also be combined if the lists have the same cardinality (number of elements), or if one of the lists has just a single element. If the two lists being operated upon have the same number of elements, then the operator is applied to the corresponding element pairs in each list to produce a new list of the same cardinality. If one of the lists has just a single element, then the operator is applied, in turn, to each of the elements of the larger list and to that single element to produce a list whose size is the same as the larger list.

If both of the elements in an operation are the same type, either integer or floating point, then the result of the operation is also of the same type. If, however, one of the elements is an integer and the other is a floating-point value, then the result is a floating-point value. Given this automatic type conversion, a simple way to convert an integer value to a floating-point value is to add 0.0 to it.

Operators

The operators listed in the following table can be used in arithmetic expressions. In the table, a horizontal line separates each precedence level and the higher levels are placed above the lower ones. Operators with higher precedence are evaluated before those with lower precedence. Operators with the same precedence are evaluated from left to right. Because expressions in parentheses are evaluated first, you can use parentheses to alter the grouping of operands, thereby changing the precedence levels.

Operator	Description
–	Unary minus
^ **	Exponentiation Exponentiation (alternate form)
* /	Multiplication Division
+ –	Addition Subtraction

The following expressions all have the value {2.0 4.0 8.0}:

```
{3.0 5.0 9.0} – {1.0 1.0 1.0}
{2 4 8} + 0.0
8 * {.25 .50 1e0}
2 ^ {1.0 2.0 3.0}
({2 2 2} – 1.0) * {2 4 8}
```

Assignment Statements

Assignment statements store values in variables. The general form of an assignment statement is:

left-side`[attribute_name:value,...] =`
right-side`[attribute_name:value,...]`

The *left-side* portion of an assignment statement is a sequence of one or more identifiers separated by commas. The `[attribute_name:value,...]` value pair lists and the brackets are optional, and are discussed in “Function Call Attributes” on page 202. The equal sign (**=**) is the assignment operator. It stores the values specified in the *right-side* portion of the statement in the variables named by the identifiers specified in the *left-side* portion of the statement. You can specify the values in the *right-side* portion of the statement as a sequence of expressions or as the result of a single function call.

You can also use either of two additional symbols, **<–** or **:=**, as the assignment operator. An assignment statement is terminated with a semicolon, indicating the end of the *right-side* portion of the statement.

The values specified to the right of the assignment operator are assigned to the identifiers to the left of the assignment operator. If the number of values equals the number of identifiers, the first value is stored in the first identifier, the second in the second, the third in the third, and so on. If the number of values is greater than the number of identifiers, the extra rightmost values are ignored. If the number of values is less than the number of identifiers, the values are assigned in order to the leftmost identifiers.

Those identifiers not receiving a value from the *right-side* list are set to the value **NULL**. In addition, if an identifier is repeated in the *left-side* list, then the *right-side* associated with that identifier is the value associated with its rightmost instance.

All identifiers that have not had a value explicitly assigned to them have the value **NULL**.

Expression Assignments

The values for the *right-side* portion of an expression assignment statement consist of a sequence of constant values, variables, arithmetic expressions, and the value **NULL**, separated by commas. The various values need not be of the same kind, data type, or dimension.

The following examples all assign the value “A string” to the variable *a*, the value 2.0 to the variable *b*, and the value **NULL** to the variable *c*.

```
a  = "A string";              // These 3 lines
b := 2 * (2 – 1e0);           // constitute a single
c <– NULL;                    // example.

a, b, c     = "A string", 2.0, NULL;
a, b, c    <– "A string", 5 / 2.5, NULL;
a, b, c     = "A string", 2.0;
a, b, c, d  = "A string", 2.0;
a, b, c, a  = [[1 0][0 1]], 2.0, NULL, "A  string",  [1 2 3];
```

The following example illustrates a simple way to swap values:

```
a = 2;
b = 4;
c = 6;
d = 8;
e = 10;
a, b = b, a;         // Values are swapped, so a = 4, b = 2
c, d, e = e, c, d;   // c = 10, d = 6, e = 8
```

Function Call Assignments

A function call can refer either to a function defined as a module (a function compiled into the system), or to a macro (a function defined in the scripting language itself). The values for the *right-side* portion of a function call assignment statement are the values returned by a single function call.

The Statistics function, which is used in the following examples, returns five values:

- Mean of the data
- Standard deviation of the data
- Variance of the data
- The minimum value in the data
- The maximum value in the data

In the first example, all of these values are assigned to variables for later use. In the second example, the minimum and maximum values are ignored. In the third example, only the minimum and maximum values are being saved for later use.

```
(1) mean, sd, var, min, max = Statistics (data);
(2) mean, sd, var = Statistics (data);
(3) min, min, min, min, max = Statistics (data);
```

10.5 Invoking Data Explorer Macros and Modules

This section describes the procedures and features for invoking macros and modules. It describes function call arguments and attributes.

Function Call Arguments

Data Explorer provides a flexible function-calling mechanism for invoking macros and modules.

A macro's definition includes a list of identifiers that are used as its input formal parameters. The formal parameters act as names and place holders for the arguments that you supply when the macro is called.

Modules (functions that are compiled into the system) have named formal parameters. The general form of a function (macro or module) call, whether used as the right hand side of an assignment statement or on its own, is:

Name (*arglist*)[attribute_name:value,...]

where *Name* is the name of the function being called and *arglist* is a list of arguments that are separated by commas (the list can be empty). Following the function may optionally be a list of attribute_name:value pairs enclosed in square brackets. Each argument's value can be either a variable identifier, a constant value, an expression, or the special identifier **NULL**. Note that nested function calls *cannot* be passed as arguments. The argument values can be passed either by position or by name, as described in the following sections.

Positional Arguments

The *positional* argument-passing mechanism is similar to the mechanism found in most programming languages that use subroutines. Given a function declared with *n* input formal parameters, the first *n* values supplied in the function call are assigned to the first *n* formal parameters. If you supply $i < n$ values in the function call, then only the first *i* of the function's formal parameters are assigned the supplied values.

The missing arguments are assigned the value of **NULL**.

By-Name Arguments

The "by name" argument-passing mechanism provides more flexibility in specifying arguments. When an argument is passed by name, the following syntax is used for the argument:

```
Fname = value
```

Fname is an identifier that corresponds to one of the function's input formal parameters. *Value* is one of the types of values that are valid for that argument to the function. If the function is a macro, the arguments are named in the definition of the macro in the script. If the function is a module, the argument names are provided in the description of the module in Chapter 2, "Functional Modules" on page 15 in *IBM Visualization Data Explorer User's Reference*.

Notes:

1. Positional arguments can be supplied only prior to by-name arguments, because the positional context is lost once a name has been supplied.
2. If an argument is supplied both by position and by name, then the value given by name takes precedence.
3. If an argument is supplied by name more than once in a given function call, then the value associated with the last (rightmost) instance of the input formal parameter is used.
4. A name that does not correspond to one of the function's formal parameters and its associated values is considered a semantic error.

Missing Arguments

Any formal parameter of a module that has not had a value passed to it, either by position or by name, is initialized to the value **NULL**. If **NULL** is explicitly passed into the module, the module may still use the default value, provided it is designed to do so. The **NULL** value allows modules to use internal defaults for those values that are not specified in a function call. The default value must be specified in the code of the module (see *IBM Visualization Data Explorer Programmer's Reference* for information).

If the function is a macro, a missing argument or an argument explicitly specified as **NULL** causes the default value to be used. If no default is specified, the parameter is set to **NULL**.

Example

The module Camera takes the following arguments:

to	The position in space to which the camera is pointed. The default is [0, 0, 0].
from	The position in space where the camera is located. The default is [0, 0, 1].
width	The width, in user units, of the camera's view. The default is 100.
resolution	The horizontal resolution, in pixels, of the image generated by the camera. The default is 640.
aspect	The aspect ratio of the image generated by the camera (i.e., its height divided by its width). The default is 0.75.

up The direction, in the world coordinate system, that the camera considers “up” The default is [0, 1, 0].

perspective The projection method. The default is 0, indicating orthographic projection.

view angle The viewing angle. This applies only in perspective projection, and the default is 30.

background The image background color. The default is “black”.

The following function calls are all equivalent and construct the default Camera Object:

```
c1 = Camera ([0, 0, 0], [0, 0, 1], 100, 640, 0.75, [0, 1, 0], 0, "black");
c2 = Camera (NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);
c3 = Camera ();
c4 = Camera (NULL, NULL, NULL, NULL, NULL, [0, 1, 0], NULL, NULL);
c5 = Camera (up = [0, 1, 0]);
c6 = Camera ([5, 5, 5], [0, 0, 1], NULL, to =  [0, 0, 0]);
c7 = Camera (width = 512, width = 640);
```

Function Call Attributes

Functions may optionally have attributes associated with each invocation of the functions, and attributes may also be associated with specific outputs of a module. The Data Explorer scripting language provides three function call attributes:

instance Identifies the instance of a function call. This can help you locate errors in scripts. To use the **instance** attribute, follow the function call with instance and the instance number separated by a colon and enclosed in brackets. For example:

```
out1 = Color(surface,"blue") [instance:1];
out2 = Color(surface,"green") [instance:2];
```

In this example, each instance of the Color module is identified uniquely. If an error occurs, the error message will report the module name, in this case Color, plus its instance number. If the instance numbering was not used, the error message will only report the name of the module.

cache Specifies whether the system writes the outputs of a module into the cache. The cache is a portion of memory in which results of previously executed functions are stored. If the inputs to a module do not change the module will not be invoked in subsequent executions, rather the module results are retrieved from the cache. If the module results are not found in the cache, because they were purged in order to make room for some other result or they were never stored in the cache, the module will be reexecuted.

The cache attribute can have one of three values:

[cache:0] - Do not cache the module outputs

[cache:1] - Cache all the results of the modules outputs

[cache:2] - Cache only the results of the modules outputs for the last execution of the module

The **cache** attribute can be associated with the individual outputs of a module, with the entire module (affecting all the outputs) or a combination of both. When output-associated and module-associated

cache attributes are used in combination the output-associated attributes override the effects of the module-associated attributes.

The following two examples illustrate how the **cache** attribute can be used. In the first example the attribute is associated with the module. In the second example the attribute is associated with the module output. The results of both of these examples are identical since Isosurface has only one output.

```
iso = Isosurface(data,value) [cache:0];

iso [cache:0] = Isosurface(data,value);
```

The following examples look similar to those above, but there is a difference because DivCurl produces two outputs. In the first example, only the last result of both outputs of DivCurl are stored in the cache. In the second example the last result of **div** is placed in the cache, but all results of curl are placed in the cache.

```
div, curl = DivCurl(data) [cache:2];

div [cache:2], curl = DivCurl(data);
```

In the following example all outputs of Statistics are not cached except for **min**. Since a **cache** attribute with a value of 2 was associated with **min**, the last result of the **min** output of Statistics will be stored in the cache.

```
mean, sd, var, min [cache:2], max = Statistics(data) [cache:0]
```

If no **cache** attribute is associated with the module, all results of the outputs will be cached unless an individual output has a **cache** attribute associated with it. The following example is similar to the one above, except that all results of module outputs are cached with the exception of min. Since a **cache** attribute with a value of 2 was associated with min only the last result of the min output of Statistics will be stored in the cache.

```
mean, sd, var, min [cache:2], max = Statistics(data);
```

group Identifies which execution group a module belongs to. This attribute is used to distribute parts of a visualization across multiple workstations. This attribute does not bind a module to a specific workstation, but identifies it to be a member of a group that may be assigned to a workstation. The assignment of an execution group to a workstation is done using the Executive module (see “Executive” on page 126 in *IBM Visualization Data Explorer User’s Reference*).

The following example illustrates how the **group** attribute can be used.

```
cwater = Import("cloudwater");
iso = Isosurface(cwater);
wind = Import("wind") [group: "group2"];
x = Compute("$0.x",wind) [group: "group2"];
mapped = Map(iso,x);
colored = Color(mapped);
camera = Camera(colored);
Display(colored,camera);
```

The second Import and the Compute will be placed into an execution group called “group2”. All other modules will be placed into one default execution group.

You can combine the various attributes for a single function call by separating them with commas, as in the following examples:

```
wind = Import("wind") [instance:2, group: "group2"];

Colored = Color(iso,"blue") [instance:1, cache:2];

Colored = Color(iso,"green") [instance:2, cache:2, group:"group"];
```

one shot Represents the script language implementation of the Reset interactor in the user interface. It sets the value of a variable to one value for the first execution and a different value (resetvalue) there after. The syntax is:

```
x[oneshot:resetvalue] = value;
```

10.6 Defining Macros

Macros are higher level processing functions that are constructed from simpler ones. A macro definition consists of two parts:

- A macro header
- A macro body

The following sections define these parts.

Macro Header

The macro header defines the macro's name, its formal parameters, and the names of values that it returns. The syntax of a macro header is

```
   macro MacroName (inputs) [ -> (outputs) ]
```

where:

- The keyword **macro** indicates that a new macro definition has started.
- *MacroName* is an identifier of the name that is being associated with the macro definition.
- The *inputs* portion of a macro header is a list of identifiers separated by commas. The list may be empty. These identifiers act as place holders for the arguments passed to the macro when it is called. If the macro does not require any arguments, then you can omit the list (but not the enclosing parentheses). The right-arrow symbol is needed only for macros with outputs. The following are examples of valid headers for macros without outputs:

  ```
  macro MyMacro(x,y)
  macro MyMacro()
  ```

  You can also specify default values for the inputs. Consider the following example:

  ```
  macro X (a = "no input", b = 4)
  {
    Echo (a, b);
  }
  ```

  The values of the arguments a and b vary, depending how the macro is invoked. For example:

```
X();                // a and b are set to the defaults, "no input" and 4
X("new value", 3);  // a is set to "new value", b is set to 3
X(NULL);            // a and b are set to the defaults, "no input" and 4
X(b = 6);           // a gets default of "no input", b is set to 6
```

See 10.5, “Invoking Data Explorer Macros and Modules” on page 200 for further explanation of the function-calling mechanism.

- The *outputs* portion of a macro’s header is a list of identifiers separated by commas.

  These identifiers act as place holders for the values returned by the macro when the macro is executed. If the macro does not return any values, then the right-arrow portion, -> (), is not necessary.

Macro Body

The macro body consists of a sequence of assignment statements and function calls surrounded by braces **{ }**. The functions referred to in these statements need not exist when a macro is defined; however, they must exist when it is executed.

Recursive and mutually recursive macro invocations are detected and prevented from executing. Statements are not guaranteed to execute in the order given in the macro’s declaration, although some partial ordering is always preserved. Calls to modules that cause external side effects (such as Display) are always executed in the order in which they were specified.

Macro Examples

The first example macro, Sum, takes two arguments. The macro computes and returns their sum.

```
macro Sum (arg1, arg2) -> (sum)
{
    sum = arg1 + arg2;
}
```

The second example macro, PrintSum, also takes two arguments and computes their sum. However, unlike the macro Sum, it does not return the computed value. Instead, it prints out using the Echo module. This example illustrates a function call (to Echo) that either does not return a value or whose return values are ignored.

```
macro PrintSum (arg1, arg2)
{
    sum = arg1 + arg2;
    Echo (sum);
}
```

The third example macro, VectorManip, implements a function to compute the cross product, dot product, and cosine of two 3-vectors. Note that the returned values do not need to be computed in the order in which they are declared.

```
macro VectorManip (vectlist1, vectlist2) -> (dot, cross, cos)
{
    cross = Compute("cross($0, $1)", vectlist1, vectlist2);
    dot = Compute("dot($0, $1)", vectlist1, vectlist2);
    cos = Compute("$0/(mag($1)/mag($2))", dot, vectlist1, vectlist2);
}
```

Note that the Data Explorer script language does not allow nested function calls. The following example illustrates a syntactically invalid function call:

```
Echo ( Sum (arg1, arg2) );
```

10.7 Using Data Explorer Script Commands

The Data Explorer scripting language provides commands to control the following aspects of the script environment:

Sequencer
These commands set up and control the Sequencer to display a series of frames.

File inclusion
This command lets you include other scripts in your program.

Prompts
These commands let you change the appearance of the prompt in the script environment.

The Data Explorer commands and parameters (if any) are complete scripting language statements and are usually terminated with a semicolon (;). There are additional commands that can be executed in the script environment by using the Executive module. See “Executive” on page 126 in *IBM Visualization Data Explorer User’s Reference*.

Sequencer

Data Explorer provides the following commands that allow you to control the Sequencer. You can use these commands in a script or by typing them to the executive.

sequence
The sequence command defines the frames that you specify in the Sequencer variables using the images supplied by a function call or expression. The following table defines the Sequencer variables:

Name	Read Only	Read/ Write	Description
@deltaframe		√	The number of steps between frames.
@endframe		√	The index of the last frame.
@frame	√		The index of the current frame.
@nextframe		√	The index of the next frame.
@startframe		√	The index of the first frame.

In the following example, the sequence command defines eleven frames for the Sequencer. These frames can be displayed using other Sequencer commands.

```
  ⋮
@startframe=0;
@endframe=10;
@nextframe=@startframe;
sequence displayobject(20*@frame);
  ⋮
```

play
This command begins execution on the frames that have been defined with the **`sequence`** command.

pause
This command stops the sequence at the current frame.

`step`
This command displays the next frame in the current sequence direction.

`stop`
This command stops the sequence display and returns to the first frame in the sequence.

`forward`
This command sets the forward direction of the sequence.

`backward`
This command sets the backward direction of the sequence.

`'palindrome on' | off`
The palindrome command with its parameters sets or unsets the palindrome mode. When you display frames in the palindrome mode, the current direction changes at the first or last frame in the series.

`'loop on' | off`
The loop command with its parameters sets or unsets the loop mode. When you display frames in the loop mode, the series of frames repeats using the settings of the forward, backward, and palindrome commands.

File Inclusion

`include`
The **`include`** command is used to interpose the contents of a file into the input stream being sent to Data Explorer. The file being included can contain both scripting language constructs and executive commands. This means an included file can, in turn, include other files.

There is currently a limit of 32 nested levels of inclusion, after which the **`include`** commands are ignored.

To include the file my.script, issue the following command:

```
include "my.script"
```

Prompts

`@prompt` and `@cprompt`
There are two at-sign (@) variables that you can set in the executive (or in a script) to customize the Data Explorer script prompt (`@prompt`) and continuation prompt (`@cprompt`). (The continuation prompt appears when you enter an incomplete command in the script environment. It indicates that you must complete the command before it can be acted upon.)

The default prompt and continuation prompt are **`dx>`** and **`>`** respectively.

The following example show how to set these variables. Note that this example shows the Data Explorer prompts as they would be displayed in the executive.

```
dx> @prompt = "DATA EXPLORER>";
DATA EXPLORER> @cprompt = "    more>";
```

If, after these commands, an incomplete statement was entered, Data Explorer would respond as follows:

```
DATA EXPLORER> a =
    more> 3 * 5;
```

10.8 Understanding the Script Execution Model

An execution model is applied to the constructs defined by the Data Explorer scripting language. This model consists of the environment structure that is maintained during function calls, the behavior associated with macro expansions, the scope rules used for locating the value associated with a variable, and the semantics associated with assignment statements and function calls.

Top-level Environment

All *global* assignment statements and *initial* function invocations occur in the top-level environment. This environment is special in that all assignment statements and function invocations initiated in this environment are always executed.

Function Execution

When either a macro or a module function is executed, a new dynamically scoped environment specific to that function call is created. Variables that correspond to the function's input and output formal parameters are created in this new environment. The variables corresponding to the output formal parameters are initialized to **NULL**. Those variables corresponding to the input formal parameters are initialized in the manner described in the preceding section. If an input and an output formal parameter both have the same name, then they share a single parameter and are initialized to the value passed as input when the function is called.

Macro Expansion

When the function being called is a macro, the macro is effectively expanded in-line after first constructing the necessary environment for its input and output parameters. This guarantees that the partial orderings defined by macros are maintained.

Variables Used in Macros

The Data Explorer allows you to use variables on both the left and right sides of a function assignment; that is, as both left-side and right-side.

Variables as Left-Side Values

All assignments in a macro's environment affect variables that are local to the macro. These variables, if they do not already exist, are created in the macro's environment when they are first used on the left-hand side of an assignment expression. Thus, a variable outside of a macro cannot be modified with that macro.

The only way to make such a change is to propagate a new value out of the macro using its output formal parameters, and to use this returned value in an assignment statement in the enclosing environment.

Given these semantics, it is possible for a local version of a variable to come into existence and obscure a more global version of a variable, midway through a macro's execution.

Variables as Right-Side Values

The values of variables used in a macro in expressions and as function arguments are found according to standard dynamic scoping rules. If the variable exists in the macro's local environment, then its value is used. Otherwise, the enclosing environments, all the way to the top-level environment, are searched to locate the variable. The value used is the value associated with the first instance of the variable. If the variable is not found in any environment, then the value **NULL** is used.

Example

The following is a sample script that illustrates how variables are treated in macros:

```
// This is a complete sample script

macro add(a, b) -> (sum)
{
c = a + b;    // c is created and given the value a+b
c = a + x;    // x is found in the top level, and used in this expression
c = a + z;    // z does not exist anywhere; NULL is used
sum = a + b;  // sum is created local to the macro, separate from the
              //     version of sum in the top level
}

x = 7;
sum = 10;
total = add(4, 4);  // total = 8, sum stays at 10
```

Assignment and Function Call Semantics

As stated earlier, all assignment statements and function calls initiated at the top-level environment are executed. In a macro, the process is slightly different. When a macro is expanded, its statements are first analyzed to determine whether they need to be executed. The rule of thumb for determining if a statement will be executed is that it must contribute, either directly or indirectly, to:

- One of the values passed into a module that has a side effect
- A value assigned to a top-level variable, using a macro's formal output parameters.

There are two kinds of “side effect” modules:

- Those that do not produce any Object outputs. Display and WriteImage are examples of this kind of module because they modify things such as monitors and files, which are outside of the language's domain and control. This type of module is represented in the graphical user interface as having no output tabs.
- Those that make use of internal executive features. These modules are specially marked in their `.mdf` (module description) files as ones that can cause side effects.

Side effect modules are always executed. (For information on the SIDE_EFFECT flag, see 10.1, “Module Description Files” on page 80 in *IBM Visualization Data Explorer Programmer’s Reference*.)

Execution Example

The macro **sum3** in the following example computes both the sum of its first two arguments and the sum of all three of its arguments. When the top-level assignment statement on line 7 is executed, the statements in lines 3 and 4 are both executed, resulting in the values 11 and 111 being assigned to the top-level variables *x* and *y* respectively. When the top-level assignment statement on line 8 is executed, only the statement on line 3 is executed. Since the value in the output variable *e* is not assigned to anything in the calling environment, the statement on line 4 need not be executed.

```
macro sum3 (a, b, c) -> (d, e)                  // 1
{                                                // 2
    d = a + b;                                   // 3
    e = d + c;                                   // 4
}                                                // 5
                                                 // 6
x, y = sum3 (1, 10, 100);                        // 7
x    = sum3 (1, 10, 100);                        // 8
```

10.9 Running .net files in script mode

When you create a visual program using the User Interface, the .net file saved is a script, so you can run it in script mode. (User Interface-specific information, such as placement of tools on the canvas, is saved as comments in the script.) If you have a sequencer in your visual program, the User Interface adds a “play” command as the last line of the .net file. Thus you can edit this line out and add your own options if you want to do something other than play forward through the sequence once (see “Sequencer” on page 206). If you do not have a sequencer in your visual program, the User Interface adds a call to “main()”, the main macro which is defined to be your top level visual program. If you do not want the program to automatically execute when you read it in as a script, remove or comment out the call to “main()”.

If your visual program uses macros, the user interface will add an “include” line so that the macros will be included when the visual program is run as a script. You can look at the top of the .net file to see which macros are referenced by the program. Thus if you need to send a collection of visual programs and macros to another person, this can help you to make sure you have sent all the necessary tools.

Appendix A. Using Data Explorer: Some Useful Hints

A.1 Using Data Explorer Effectively

Following are some hints for using Data Explorer more effectively, debugging visual programs, and using memory efficiently.

Common Problems

Debugging

One of the most useful tools for debugging visual programs is the Print module. For example, if you are getting an error from a module that a particular field is inappropriate for processing, you can print out the object to see if it is what you expect it to be. Print can be used to see the structure of and data values in any object. The options parameter is used to set the level of detail printed: the default "o" prints just the top level object: for example "Field with 4 (four) components." If options is set to "r," more information about each component is printed, for example, how many items in each component, and the data type. You can also print out some or all of the values in the components.

The output of Print appears in the Message window.

Stopping Execution

There are two ways to stop execution of a visual program:

- End Execution in the Execute menu stops execution *after* the currently executing module has finished.
- Disconnect from Server in the Connection menu kills the Data Explorer executive (not the user interface) immediately. You can restart, using **`Start Server`** in the **`Connection`** menu.

In addition, modifying a visual program (for example by disconnecting an arc or adding a new tool) will cause execution to stop after the currently executing module.

How to orient yourself in the Image window

If you find yourself "lost" in the Image window; for example, you have a black picture and don't know where your data object is, you can always "reset the camera" by using the Reset Camera option in the View Control dialog box of the Image window. This zooms out so that you can see *all* of your object, from a "front and center" view.

It is also often helpful to use ShowBox to display the bounding box of your entire data set. Collect this with the rest of your visualization, and then you will be able to see how the part you are looking at relates to the entire data set.

What is the Difference Between Image and Display?

Image, Display, and Render all render an object (i.e. create an image).

Render, given an object and a camera, creates as output an image. This image can be sent directly to Display for display to the screen, sent to WriteImage to be written to a file, or collected with other images into a single window using Arrange.

Display, given an object and a camera, both renders the object (using the camera) and displays it to the screen.

Display, given only an image, simply displays it to the screen.

Image, given an object, renders it and displays it to the screen. The camera information is provided via direct interactors (rotate, zoom, etc.) or through the camera mode option in the View Control dialog box. Image has two outputs: the object to be rendered (including any AutoAxes that may have been added via menu choices) and the camera used.

You would use Render if you needed the image itself, for example, for the Arrange or Filter modules, or if you wanted to use WriteImage. (For the Image tool, the WriteImage function is available through the Save Image and Print Image commands in the Image window, or through the hidden recordEnable, recordFormat, and recordFile parameters to the Image tool).

You would use Display without a camera if your object is already an image, and you simply want to display it. You do not need (or want) to render it. You would also use Display without a camera to display a set of Arranged images.

You would use Display with a camera if you wanted to directly control the camera, for example, for a computed fly-through path. You would also use Display if you wanted to define your own direct interaction modes (see “SuperviseWindow” on page 336 and “SuperviseState” on page 332 in *IBM Visualization Data Explorer User’s Reference*), rather than using the predefined direct interaction modes of the Image tool.

How do I get more information?

- In the Help Menu:
  - **`Application Comment`** presents comments (if provided) on the current visual program.
  - **`Table of Contents`** presents the table of contents of the user documentation. You can use hypertext links to go to a particular topic.
  - **`Context-Sensitive Help`** presents a “?” cursor: just click on a tool icon or other feature to learn more.
- Samples:
  - Sample Visual Programs: /usr/lpp/dx/samples/programs. See also the subdirectories there, grouped by topic.
  - Sample Scripts: /usr/lpp/dx/samples/scripts.
- Information available electronically:
  - Data Explorer user group on the internet (`comp.graphics.apps.data-explorer`)
  - DX Home Page on world wide web (`http://www.almaden.ibm.com/dx/`).
  - Data Explorer Repository at Cornell: anonymous ftp: `ftp.tc.cornell.edu.` (look for directory `pub/Data.Explorer`).
  - gopher: `ftp.tc.cornell.edu.` port 70.

Memory Use

Data Explorer Object Cache

Data Explorer uses an object cache to store intermediate results of modules. Caching systems are intended to fill up and then reclaim memory by throwing things out of the cache. The size of the cache defaults to a large percentage of the physical memory on the machine. You can control the size of the cache with the `-memory` command line option to the **`dx`** command. The minimum cache size needed is on the order of the maximum amount of memory required for a program execution.

The Data Explorer "executive" schedules module execution. It does detailed graph analysis, implements distributed processing of the modules, and implements the Switch and Route modules. It also provides optimization by caching the intermediate outputs of modules. For example, if you run Import twice in a row with the same inputs, Import will not actually run the second time, and instead the executive will use the cached output from the previous execution. The Image and Display tools also cache their images internally.

To implement the caching scheme, Data Explorer will allocate memory up to some fixed size. This memory is referred to as the arena. When the arena fills up and more memory is required, Data Explorer looks for objects to discard from the cache. When it does this it may mean that subsequent executions will have to execute larger portions of the program.

The arena is of fixed size for any one instance of Data Explorer. The size of this arena is chosen by default based on the size of the physical memory in the system.

For some data sets, the default arena size will not be sufficient. In those cases, one can use the `-memory` option to increase the size of the arena, with the limitation that your can't increase the arena size to be larger than the amount of real plus virtual memory (page or swap space) on your machine. Talk to your system administrator if you think you need to increase the amount of swap space on your system.

Reducing Memory Requirements

If, after using the `-memory` option as described above, you find you still lack sufficient memory to perform your visualization, there are a number of strategies that can be used to reduce the amount of memory that is required by your program.

Do Not Render Images: A common mistake is to render image data (i.e. 2-dimensional grids) using Render, Image, or Display with a camera input. This results in Data Explorer interpreting the image as a very large number of quads, in which case much memory and CPU is used.

Instead, one can AutoColor or Color the image and pass it directly to Display (without a camera input), or for even more memory savings, convert the data to unsigned bytes (see below) and AutoColor or Color the data with delayed colors (see below).

Delayed Colors: If you are coloring your objects (using AutoColor, AutoGrayScale, or Colormap/Color), you might want to use "delayed" colors.

To do this, convert the data component to unsigned bytes and set the "delayed" parameter of the coloring module to 1. Using delayed colors means that rather than a 3-vector being used for each data point, a single scalar byte is used to index into a color table with 256 entries.

If you are using ReadImage, you may want to set the DXDELAYEDCOLORS environment variable. See “ReadImage” on page 250 in *IBM Visualization Data Explorer User’s Reference*.

Converting Data Types: In many cases it may be acceptable to convert your data components to smaller sized types using Compute. For example, you might change your floating point data to bytes. This has the advantage that all downstream modules will require less memory.

Working with Series Data: When working with series data, if you are importing the entire series and then selecting members out of the series, it may be that your program can be changed so that you only import one member at a time. Do this using the **start** and **end** parameters to Import.

This reduces memory requirements by not having the whole series in memory at once.

Glyphs: If you are using glyphs (AutoGlyph or Glyph), you may want to use less “spiffy” glyphs. A less spiffy glyph is one that has fewer positions and connections (facets), and therefore consumes less memory. To use less spiffy glyphs, use the **type** parameter of either AutoGlyph or Glyph, and set it to “speedy” or to a small fraction of 1.

Reducing Grid Resolution: If you can sacrifice resolution in your data set, you may want to use the Reduce module (usually just after Import) to reduce the number of points in your data set. Reduce filters the data set before reducing the number of points. Remember that it is of little use to process 5000x5000 points if your final image is only 1000x1000 pixels.

24-Bit Images: You can create 24-bit images (instead of the default 96-bit images) by setting the environment variable DXPIXELTYPE to DXByte. See “ReadImage” on page 250 and “Render” on page 264 in *IBM Visualization Data Explorer User’s Reference*.

Cache Control: Executive: In general, it is not necessary to change how the executive caches intermediate results. However, in a few cases, it may be advantageous to do so. For example, if you are reading a live data feed into your program, it is probably not necessary to cache the downstream outputs.

You can change how and if the executive caches intermediate output values by opening the Configuration dialog box of a module and changing the option menu to the right of each output. You can also choose **Output Cacheability** from the **Edit** menu of the VPE, and set the cacheability of a group of modules, show the cacheability of a group of modules, or ask Data Explorer to use a heuristic to automatically optimize the caching for the current visual program.

In general, it is most efficient to cache only the results of the last module in a single file line of modules; for example to cache the output of Isosurface, but not Import. Note that if you do this, however, if you need to change the isosurface value, the data file will need to be reimported, slowing execution.

If you want to turn off caching altogether you can use the `-cache off` command-line option to Data Explorer.

Cache Control: Display: Some modules use the caching system to cache their own data. The Display and Image tools are such tools. When using software rendering, they cache the images they display in the X windows. This is an optimization that can be seen when using the Sequencer. When this tool starts repeating itself (in loop or palindrome mode), the images are displayed much faster. That is, Display (or Image) is pulling them out of the cache instead of rerendering the input objects each time. You can observe this effect by running the example program MovingCamera.net with software rendering.

Most of the time this caching behavior is desirable, but in some cases it is better turned off. To do that, use the Options module to add a "cache" attribute with the integer value of 0 (zero), as follows:

```
o = Options(o, "cache", 0);
Display(o, camera);
```

The Image tool's Configuration dialog box has an **`option`** menu that lets you control its caching. This can be useful when one is running a batch job to generate an animation in which none of the frames will be displayed a second time.

Note that the `-cache off` command line option mentioned above has no effect on the internal caching that modules themselves perform.

Note: You can use the Data Explorer command line option `-optimize memory`, which will automatically set the DXDELAYEDCOLORS and DXPIXELTYPE environment variable to the options that consume the least memory. The alternative is `-optimize precision`.

System Tuning

Default Memory Size: Except where noted in the architecture-specific README (in /usr/lpp/dx), by default Data Explorer will be allowed to grow to use all but 8 megabytes of the physical memory when there is less than 64 megabytes of physical memory.

If there are more than 64 megabytes of physical memory, then Data Explorer will, by default, be allowed to grow to 7/8 of the amount of physical memory.

Users may wish to alter this default amount of memory by using the `-memory` option to the **`dx`** command, or the "Memory" field of the **`Connect to Server Options`** dialog box.

Paging Space: Since it is possible for Data Explorer to use a large amount of virtual memory, users should configure systems with paging space at least two or three times the total physical memory in their system.

If you do not have enough paging space, the operating system may kill Data Explorer (or other processes), sometimes without warning, depending on the architecture. Your system administrator can increase your paging space.

Per Process Limits: Some systems may enforce per process limits on such things as data segment size, stack size and so forth. These may need to be adjusted to run Data Explorer with large amounts of memory to avoid paging. Your system administrator can adjust your per-process limits.

A.2 Visualization Techniques

Now we have our data in a Field Object inside Data Explorer. What can we do with it? This section discusses some common visualization techniques and the Data Explorer modules associated with them:

- “Animation”
- “Annotation” on page 218
- “Color Mapping” on page 219
- “Contours and Isosurfaces” on page 221
- “Mapping” on page 223
- “Normals and Shading” on page 224
- “Plots and Histograms” on page 227
- “Rubbersheet” on page 227
- “Transformations and Structuring” on page 228
- “Vector Fields” on page 229
- “Volume Rendering” on page 231.

Animation

The Sequencer tool is the primary device used in Data Explorer to produce animation or motion control. There are two basic types of animation: show a series of steps one after another, or, cause an object to move or rotate or change scale in order to study it from different points of view.

Since many data sets are measured at a series of different times, your data may have a “time value” associated with each measurement set. There are two ways to read in these time step data files in order to study the dynamic process you have measured.

In one scheme, you can collect all your data files into a *Series*, a special Group of Fields understood by Data Explorer. Each Series member can represent a data collection event at a certain time. Series do not have to be based on time; you may have a set of experimental measurements made at different voltages (e.g., a voltage series). Each series member is assumed to have the same type (scalar, vector, etc.) and the same dimensionality (2-D, 3-D, etc.), but the data and even the grid size or number of connections and positions may be different for each Series member. The Series Field is described in detail in “Series Groups” on page 35. Series “values” do not have to be continuous but may represent useful information like the actual voltage setting for that Series entry (0.04, 2.3, 13.4). Series members are accessed by their ordinal position, starting at 0, regardless of their “value.”

Another way to organize a collection of associated data files is to create individual files for each time step (or voltage measurement, etc.). Give each file a filename containing an ordinal number so you can access them easily with a computer program (e.g., myfield.001.dx, myfield.002.dx, and so on). Each file will contain the Field to be imported at each time step.

In either case (Series Field or separately numbered files), you can control when a particular time step is visualized in a visual program using the Sequencer tool. This tool emits a series of integers. You set the minimum, maximum, and increment, as well as choosing to start at a specific number (so you can jump ahead in the series if you like). The Sequencer can be connected to the Import module to specify which Series member to read in from the specified input file at the next iteration.

Alternately, you could Import the Series Group file with Import, then use the Select module. Select takes an integer input (from Sequencer, for example) to choose the appropriate series member.

If you choose to use separate files for separate data samples, you would likely want to use the Sequencer as an input to a Format module. The Format module could construct the filename with a format string like `%s%03d%s` along with three inputs, "myfield.", the output of the Sequencer, and ".dx". Then, when the Sequencer emits the integer "2", the output string from Format becomes "myfield.002.dx". This can be fed into Import as the name of the .dx file to read. The result is that you can use the Sequencer to specify either any specific file or a whole series of files to import and image one after another.

Another common type of animation is to use the Sequencer to control object motion. Usually, this requires that you run the output of Sequencer through at least one Compute module. For instance, you can rotate an object around the Y-axis one full revolution by employing the Rotate module. The smaller the angular increment, the smoother the animation will appear, but there is a trade-off in apparent motion rate if your graphics workstation is not very fast. So you may have to adjust the incremental angular amount to your liking.

You will find the technique of wiring Interactors to Compute modules useful for converting the output of Sequencer to arbitrary floating-point values. If you wanted to vary the Scale of your object using the Scale module, it might be more convenient to adjust the scale in increments of 0.01. With a little thought, you can extend this idea so that the same (one and only) Sequencer integer series can be converted into several different series of numbers that can simultaneously rotate, scale, and read in different time steps of data. Just a caution, though: too much changing at the same time will probably not help you visualize your data, but instead will cause confusion. Is the object getting bigger because the data values are increasing, or because you are changing the scale, or because you are moving the object closer to you with Translate? When you start out, keep your animations simple and they will be much more effective.

Annotation

It is imperative that good visualizations contain sufficient annotation for a viewer to derive appropriate information from the imagery. A colored height field or streamline set with no supporting labeling can make perfectly beautiful, utterly meaningless computer graphics.

Annotating a scene can be done in several ways using Data Explorer modules. You can, for example, provide a ColorBar with numeric values automatically labeled next to the related colors, show Text or Caption information to provide textual descriptions of objects, or turn on AutoAxes to show neatly labeled and numbered axes around the perimeter of your data space.

Using the Format module, it is possible to create "clocks" or other "meters." Format creates a formatted string of text suitable for Caption or Text modules to display. Format takes a "template" and text strings and/or numbers as **`value`** inputs and assembles an informative text string as output. For example, inputting the minimum value of your data to the first **`value`** input (the second input tab) of a Format module, you could create a Caption that reads:

```
Minimum temperature = 0.0 deg.
```

To do this, the “template” inside the Format module would read:

```
Minimum temperature = %1.1f deg.
```

In this template, the “%1.1f” serves as a place holder for the first value (which must be floating point) provided to Format; consequently, the minimum value argument is substituted into the string when the visual program is executed. The “1.1” means that the floating-point number should display at least one number to the left of the decimal point but should round off to only one decimal place to the right of the decimal. By tying the data Field to Statistics (Transformation category), you can easily extract the minimum value of the data; use this as the second input to Format. If you later input a different data set with a different minimum, Caption will automatically change to reflect the new minimum value.

One trick for showing text together with numbers that are changing is to use a “fixed width” font instead of a “variable” or “proportional” font. Variable text looks better when making Captions that do not include changing values, but fixed width text maintains the same width regardless of the numeric characters currently being displayed. Try both ways and you will see that the variable text has an annoying shrinking-expanding effect as your clock or time step meter changes value. To get the fixed text clock to behave correctly, you must use a Format template like “%03.2f” that allows for enough numbers to the left of the decimal point. In this example, we have predetermined that we will never create a number greater than 999.99 (note that if we *do* go over 1000, the text will expand to show the whole number, causing the Caption string to expand: the very thing we are trying to avoid!). The “%03.2f” format makes floating-point numbers with 3 numerals before the decimal, including left side zero padding, and 2 numerals after the decimal.

Color Mapping

Data Explorer provides an automatically generated color map (AutoColor), an automatically generated grayscale (AutoGrayScale), and a user-definable color map (the Colormap module that attaches to the Color module). A color map represents a relationship between a continuous range of floating point numeric data values and a set of color values. Frequently, you will encounter color maps with continuous (“spectral”) color tones like a rainbow, but there is no requirement that color maps appear continuous. Each color map has associated with it a minimum and a maximum scalar value. You can either specify the minimum and maximum or connect the data Field to the Colormap module and have these values automatically extracted.

We can describe “color” to a computer in a number of ways. One of the more intuitive is the “hue-saturation-value” model used by Data Explorer's Colormap tool. Hue is the color’s “name”, like blue, red, and so on. Hue is considered to form a circle from red through yellow, green, cyan, blue, magenta, and back to red; think of Hue as an “angle” around this color wheel (scaled from 0.0 to 1.0). Saturation is the “richness” of a color. Decreasing the Saturation of a color from 1.0 to 0.0 makes the color progressively more pastel, so for example, bright red becomes light red, then pink, finally turning white. You can think of decreasing the Saturation as adding “white paint” to paint of a pure hue. At Saturation 0.0, any color becomes white (assuming Value is held at 1.0). Similarly, Value is a measure of the amount of “black paint” mixed with a color. As you decrease the color’s Value from 1.0 to 0.0, you add more “black”, so bright red becomes progressively darker red, and finally black. Any color becomes black at a Value of 0.0. All three of these parameters interact, so you can adjust Hue and decrease Saturation and Value to get a “dark pastel blue.”

Another scheme for describing color is RGB (Red-Green-Blue). As in the HSV model just described, you specify a color as a triplet (a 3-vector). Each component can have a value from 0.0 to 1.0. If all three are 0.0, the resulting color is black; if all three are 1.0, you get white. Given Red = 1.0, Green = 0.0, Blue = 0.0, the color is fully saturated bright red. You can observe a graph of RGB lines at the far left of the Colormap tool as you manipulate the colors using the Hue-Saturation-Value (HSV) controls. You can specify an RGB vector in the Color module in place of connecting a Colormap if you want the output object to have a single color (or you can specify one of the X Window System color names). And you can convert from RGB to HSV or back using the Convert module. See “Color” on page 75 and “Colormap” on page 84 in *IBM Visualization Data Explorer User’s Reference* for more details about these different specification schemes.

Let us assume that we have set the Colormap minimum and maximum to equal the minimum and maximum of the temperature data we collected in the atmosphere (this is done automatically if you connect the data Field to the input on Colormap). Recall that we collected position-dependent data, one temperature value at each grid position. For this example, assume the minimum temperature measured was 0 degrees Centigrade and the maximum 20. What color is 10? That depends entirely on the color map used. If we have a standard spectral (rainbow) map with blue at 0 and red at 20, then 10 would have a color halfway between blue and red. On the default color map, this would be green. When we ask Data Explorer to color-map our data, it examines each data value, performs a linear interpolation between the minimum and maximum values to find the color associated with that interpolated value in the color map and “colorizes” the object at all points containing that data value with that color.

If we change the maximum value in the color map to 30, the measured data value of 10 (taken from the same data set as above) will now map to a cyan color, part way between blue and green. On the other hand, we could keep our same extreme values but manipulate the color map’s color distribution in such a way that any value has any color we like. You can learn the details about this capability in 6.3, “Using the Colormap Editor” on page 119.

The best way to learn about the power of color mapping is to take some sample data, color-map it, then manipulate the settings in the Colormap Editor you have connected to the Color module your data Field passes through.

Note: Choose **`Execute on Change`** from the Colormap Editor **`Execute`** menu and you will see the data change colors as soon as you make a change in the Colormap Editor.)

For instance, you can create sharp color discontinuities by placing two control points close together vertically on the Hue control line, then dragging one horizontally away from the other. This can be used to indicate a sharp edge transition in your data. It is sometimes useful to place a special contrasting color in the middle of an otherwise continuous color map. For example, to highlight the value of 12 degrees C in our temperature data, we could insert a sharply defined red notch or band into the middle of our smooth rainbow color map. This would highlight that particular value or range for someone examining the scene. You can automatically generate a number of control point patterns by choosing **`Generate Waveforms...`** from the **`Edit`** menu in the Colormap window. To make a notch, choose one of the "S" shaped curves from the pop-up menu in the **Generate Waveforms** dialog box. Set the number of Steps to 4 to make a single notch, or 3 to make a single step. Click **Apply** to place control points on the currently chosen curve (Hue, Saturation, Value, or Opacity). You can then drag the new control points where you like.

If you use a red color notch in the middle of your data range, you probably will not want to use red elsewhere in your color map or it will be difficult for a viewer to tell the 12-degree specially highlighted red area from the 20-degree red maximum values (assuming 20 is the maximum). In fact, it might be safer to use a white or gray color to mark the special value of 12 degrees. Do this by creating a notch on the Saturation or Value curves instead of on the Hue curve.

Similarly, you can change the *opacity* of objects. Opacity is the inverse of transparency: that is, the more opaque the object, the less transparent. You can set opacity to a value between 0.0 and 1.0. Opacities less than 1.0 allow you to see through an object to reveal objects inside or behind the transparent object. For all objects except volumes, an Opacity of 0.0 will make the object disappear completely. Since Data Explorer uses an emissive volume rendering technique, you must set the color of a volume to "black" (RGB of [0, 0, 0]), as well as setting the Opacity to 0.0, to make the volume disappear. You will notice that when you view slightly transparent objects through each other, the colors of each object combine, making it very difficult to accurately assess the color of any one object. Used sparingly, opacity is a very powerful tool for examining the insides of objects or volumes and gauging the physical relationships between intersecting objects.

You can create a variable opacity on an object by manipulating the opacity curve in Colormap. This can make parts of an object trail off to transparency, useful if some data values are not of interest. Be aware that "hiding" data in this way may mislead someone viewing your results. But in some data sets, there may be a large number of "noisy" data values that you would like to exclude in order to see the "signal" data values of interest. In that case, setting an Opacity notch to hide the noisy values may be the best visualization technique.

When you lower opacity below 1.0, you will see two stripes, one white and one black, or a checkerboard pattern of black and white behind the sample color strip in the Colormap Editor. These are useful when you manipulate Opacity to check the apparent color against both a light and dark background. As with the color tools, you can turn on **Execute On Change** and interactively play with the Opacity of the selected object until you get the effect you want.

Contours and Isosurfaces

Given a set of samples taken over a presumably continuous region, it is meaningful to consider drawing smooth lines connecting together the locations on the grid containing the same data values. You are probably familiar with topographic maps that show contour lines connecting together the same values of elevation of the Earth's surface features, such as hills and valleys. These lines are called "contour lines" or "isolines" (*iso* means "same" or "equal"). In most cases, the places on the surface of the sample grid that have identical data values will not coincide with the grid sample points. This is another case where the "connections" component is required for Data Explorer to determine where on the grid the same value occurs (say the value 5.2) in order to create lines connecting together all these locations.

To return to our 3-dimensional data set taken from the atmosphere. Since we have collected data throughout a 3-dimensional space, we can identify volumetric elements defined by connecting adjacent grid sample points in three dimensions using a "connections" component like cubes. It now becomes possible to draw "isosurfaces" rather than "isolines." An *isosurface* is that surface cutting through a volume on which all data values are equal to a specified value. Depending on the actual distribution of the data, isosurfaces may look more or less like flat sheets (the isosurface of "sea level" in a data set of elevations would look like this); it might enclose a portion of our space or appear as a whole set of small disconnected surfaces or enclosed spaces.

To create an isosurface, we pick a value of interest. Suppose that according to our knowledge of meteorology, we know that the dew point (at which water condenses from vapor to liquid) is 12 degrees C in our sample. Although we measured temperatures at only a fixed number of grid points, we are interested in seeing where rain formation may begin throughout the atmosphere. We could show only the sample points highlighted by themselves, but once again, we make a reasonable assumption that we have taken discrete samples from a continuous natural volume. In other words, rain formation will not simply occur at the limited set of discrete points where we have sampled temperatures of 12 degrees C, but at all the points in between that are also at 12 degrees. How do we find all those in-between points? By interpolating through the volumetric elements between adjacent sample points. And in fact, the Isosurface module will do this automatically.

The resulting isosurface will represent all values of 12 degrees C throughout our volume of sampled space. The actual image depends on the distribution of the data, of course. If the outside of a rain cloud were at exactly 12 degrees C, we would see a shape resembling a cloud in the sky. But if rain formed at an altitude where the temperature was 12 degrees C, we would instead expect to see a flat sheet. Or we may not know what to expect: that is one of the uses of visualization, as well—for discovery, not just for verification.

Generally, the vertices that describe the mesh positions of an isosurface will *not* coincide with the original grid points. It is important to realize that an Isosurface is a new and valid Data Explorer Field with positions and connections and a data component (in which all data values are identical). You can treat this Field just like any data Field you have imported. Color mapping such a Field is not particularly useful since all the data values are identical, so you will get the same color for every point.

To draw contour lines on a 2-dimensional grid, you also use the Isosurface module. Data Explorer figures out the dimensionality of the visualization by looking at the input data. Thus, a biologist's 2-D grid can be easily contour-mapped with the same tool as a meteorologist's 3-D volume, but the visual output will be appropriately different for the different inputs. Similar to Isosurface's contour lines is the output of the Band module. This yields filled regions between contours; these bands can be colored by a color map or AutoColor to yield the kind of image frequently used to show temperature distributions on a weather map.

Mapping

There is a very useful module called Map in Data Explorer that permits you to "map" one data set onto a Field defined by another data set. For example, in our rain cloud data, we have measured temperature and cloud-water density throughout a volume. We learned earlier how to make an isosurface of temperature equal to 12 degrees C. Now it may be instructive to observe the cloud-water density associated with this temperature isosurface.

The operation we wish to perform is to use our temperature isosurface with its arbitrary (data-defined) shape as a sampling surface to pick out the values of cloudwater density as they occur throughout the volume. That is, conceptually, we will *dip* the temperature isosurface into the cloudwater volume. Wherever the isosurface comes in contact with the cloudwater volume, the values that *stick* to the isosurface represent the values of cloudwater density that occur at that intersection. But remember that the isosurface was created using temperature data. The isosurface of temperature (the *input* Field to Map in this example) had only one data value (12 degrees C) at every position, but the mapped isosurface (the output of Map) will contain arbitrary patches of data corresponding to the distribution of cloudwater density. If we AutoColor this output isosurface, we will see an arbitrary geometric surface with a patchy color scheme. The surface is the location of all 12 degree temperatures, and the patchy color corresponds to the distribution of different cloudwater densities sampled on that surface. (Of course, if cloudwater density happened to have the same value at all points on the 12-degree temperature surface, we would see only one color.)

Naturally, you can do the opposite! First, make an isosurface of cloudwater density, say at the mean value of density. The mean value of a Field is taken as the default value by the Isosurface module: this is convenient when you start exploring a new data set and do not know what the extreme values are. Now map the temperature data onto the cloudwater isosurface. Run the output through AutoColor. The result will look very different. This time, you have "dipped" the cloudwater isosurface into a "bucket" of temperature data. Once again, this serves as a reminder that you must indicate to an observer exactly what kind of operation you performed if your visualization is to bear any meaning.

You can also dip the cloudwater isosurface into the temperature *colors*. To do this, first AutoColor the temperature data set. Then use Mark to "mark" the colors as data (this temporarily renames the colors component to data, while saving the original data component). Then use Map to map this marked Field into the cloudwater isosurface colors component. (It is necessary to mark the colors as data before mapping because Map always maps from the data component). An example visual program that performs each of these mapping operations can be found in **`/usr/lpp/dx/samples/programs/UsingMap.net`**.

Note that we changed the order of the modules slightly in the third example. In the second case, we Mapped data values from the "map" Field (cloudwater density) onto the "input" Field (the temperature isosurface), then AutoColored the resulting Field. In the third case, we AutoColored the "map" Field (temperature), then mapped color values onto the "input" Field (cloudwater density). This illustrates some of the flexibility of both the Map module itself and Data Explorer in general. In this case, the output image would be similar whether you colored by temperature then mapped, or mapped temperature first, then colored by temperature. There will be color differences if the range of values that mapped onto the isosurface is different from the entire data range used to AutoColor the entire temperature Field.

You could avoid this problem by substituting a Color and Colormap pair in place of AutoColor, then connecting the original temperature Field to the input of the Colormap. This would automatically lock the minimum and maximum to the entire range of temperature, not just to the range of values that happened to fall on the isosurface.

But there are other cases in which commutative ordering of modules will yield a quite different visual output. For example, suppose we have a volumetric Field containing both vector data and a scalar data set. We can generate a series of Streamlines through the vector Field, Map the scalar data from the volume through which the Streamlines pass onto these lines, then AutoColor the lines according to the scalar data. To make the lines easier to see, we employ the Tube module to create cylinders along the path of each streamline. The radius of the Tubes can be adjusted until we get the look we like. By performing the operations in that order, the original colors are carried from the lines out to the outside of the cylinders, resulting in distinct circumferential bands of color on the Tube surfaces.

Now, change the order: create Streamlines, then Tube the lines. This yields uncolored cylinders. At this point, we Map the scalar data values from the volumetric Field in which the cylinders are embedded onto the surfaces of the cylinders, then AutoColor. This time, we will have patches of color on the cylinders, since it is highly unlikely that the volumetric data would lie in perfect rings around the outside of the tubes.

Which of the above two representations is “correct”? Both are accurate. Which you choose to show depends on the point you are trying to make. In the first case, you are illustrating the values of data precisely as they occur along the Streamlines: the Tubes are used to make these very thin lines more visible. In the second case, you wish to sample the data volume at a specified radius away from a given Streamline. By varying the radius of the Tubes, you can investigate phenomena such as the rate of change of the data Field as you move further away from the Streamline itself.

Normals and Shading

Another Field component used in Data Explorer is the “normals” component. *normals* are unit vectors that tell the computer graphics program and the image renderer which direction is “up” or “out.” Several tools, like Isosurface, automatically create a normals component so you do not have to calculate these numbers yourself.

There are two types of normals provided in Data Explorer, “connections normals” and “positions normals”. Connection-based normals are vectors perpendicular to each connection element on the surface. They are created by the Normals module when you set the **method** input to “connections”. The resulting surface reveals the underlying polygonal grid structure of your sample grid. Frequently, this is a valuable way to show your data, as any observer can then see the grid resolution directly. At the same time, this surface can be colored or color mapped either by connection-dependent or position-dependent data.

The other type of normals are created by the Normals module when you enter “positions” as the **method** (this is the default method, in fact). In this case, the surface will be much smoother in appearance yielding a more aesthetically pleasing surface at the expense of being able to directly perceive the grid resolution. It is sometimes less confusing to use position normals in place of connection normals because the object is less "busy" looking. You must be the judge of what is the appropriate way to observe your own data. You can also show your data first with connection normals, to illustrate the sample resolution, then switch to position normals in order to better show some other aspect of your data.

Normals are used by various modules in Data Explorer. One use of this information is that it is required by the image renderer (the Image, Render, and Display modules all incorporate the image renderer) to calculate the amount and direction of light falling on an object's surface (we will discuss this in more detail below). Rubbersheet assumes that the input grid or line is flat (if there is no "normals" component in the input Field) and projects the values in a perpendicular direction. However, you may wish to create your own normals or modify an existing "normals" component (using the Compute module, for example) and Rubbersheet will then use the modified normals to control the direction of projection of the surface or line. After performing the Rubbersheet projection, you may want to insert another Normals module. This will take the projected object and generate real surface normals before rendering, resulting in better-looking shading on you projected surface. See "RubberSheet" on page 277 in *IBM Visualization Data Explorer User's Reference* for a full description.

Isosurface will also generate normals automatically; to do so, Isosurface either calculates or reads the previously calculated Field gradient (depending on the setting of the **gradient** input flag). Therefore, the normals generated by Isosurface are not necessarily perpendicular to the connection elements generated by the Isosurface module, but better indicate the actual Field direction than simple perpendicular normals.

If you wish to understand Normals better, you can use the Glyph module to visualize them. First use Mark to mark the "normals" component. This makes Data Explorer treat the "normals" component as if it were the "data" component. Then, Glyph the Field. Finally, Unmark the **normals** to restore the previous **data** component to its proper place. By showing the normals as vector glyphs in conjunction with a surface, you should be able to see how different modules, like Rubbersheet and Isosurface, deal with these vectors.

Normals are also useful in helping you determine the "inside" and "outside" of an object. In addition to a "colors" component, which holds the color-mapped information for each data point in a Field, you can specify a "front colors" and a "back colors" component. Which is front and which is back is determined by the direction of the normal for that vertex (position normals) or polygon (connection normals). By setting different colors for the inside and outside of a complicated object, you may be able to understand its shape better. This technique can also be helpful when you are trying to convert a connection list like a finite element mesh into Data Explorer form. If you accidentally describe the "winding" (rhymes with "binding") of a polygonal face in the wrong order, the normal for that face will point in the wrong direction. Setting "back colors" to red and "front colors" to white will clearly indicate which faces are pointing the wrong way.

The Shade module employs the "normals" component; it will make a "normals" component if it does not already exist. Shade allows you to set up the lighting of your objects to make them more "realistic" in appearance. That is to say, when we observe a 3-dimensional object, the way light falls on the object is an important cue to our eyes that helps us understand the shape of the object. We expect the surfaces of the object that are generally facing a light source to be brighter than those that face away. Data Explorer, like other computer graphics rendering programs, takes the normal directions of the object surfaces into account when calculating the angle between the object, the light(s) in the scene, and the viewer's eye point (the camera in the scene).

In the real world, different materials react to incident light differently. For example, many metals scatter light causing the "specular" reflection to be more spread out than it is on shiny plastic surfaces. The specular highlight is the highlight (many types of cloth and other dull surfaces have no specular brightest spot on a shiny surface. Think of how the sun sometimes bounces off the hood of your car at just the right angle and makes a bright sharp reflection. By adjusting the "specular" and "shininess" inputs to the Shade module, you can make your object appear more metallic or more plastic. If you turn the specular value all the way to 0.0, you eliminate the specular reflection). This can be important if you are trying to make sense of color-mapped data, since the specular highlight will be a bright white area on the surface of the object (assuming the incident light color is white). This white spot or area could confuse a viewer who is trying to interpret the color mapping of the data.

Two other inputs in the Shade module (**diffuse** and **ambient**) are also used by Data Explorer when it lights an object. Diffuse light is light emanating from a direct light source, like the default Light in any Data Explorer network, or from Light modules you place in your network. Think of diffuse light as the light coming from a light bulb and falling on an object surface, like a light in your office shining directly on your desk. This property is called "diffuse" because it represents the way light bounces off a surface, depending on the "roughness" of the surface. The rougher the surface, the more the light rays are scattered ("diffused"). An extremely smooth surface tends to bounce light more uniformly to the eye. Ambient light is light that is indirect: for example, daylight coming through a window, bouncing off white walls and then impinging on your desk. Data Explorer automatically places an AmbientLight value in any scene, or you can override this value by placing your own AmbientLight module in a network. Ambient light is best thought of as a sort of "glow" emanating from a non-point source of light and therefore illuminating even the parts of objects that face away from the point light sources in a scene. If you remove the ambient light, the apparent "shadows" on an object lit only by a point source of light are much harsher.

Like Normals, the Shade module can light an object in two fundamentally different ways. If you enter "smooth" in the **how** input to Shade, the surface will appear smoothly rounded (assuming it is not completely flat to start with). This is equivalent to setting "positions" in a Normals module. Shade will, if necessary, create position normals, then light the object accordingly. Any point on a connection between positions will be lit by calculating an interpolated normal value between the position normals. If you choose **faceted** in Shade, the effect is the same as selecting "connections" in the Normals module. In this case, each connection element has one normal direction over the entire face. As a result, every point on a connection element reflects light exactly the same way. The image that you see will thus show faceted polygons. Once again, while this may make the object look less "realistic," it does more accurately reflect the sampling resolution of your data and may therefore be a more desirable image to show other viewers.

Plots and Histograms

Data Explorer provides a Plot module that will give you a simple 2-D graphics plot of your data. This can be convenient for showing one parameter plotted “traditionally” while you show a colored 3-D height Field illustrating the same or other parameters, in the same scene.

Histogram regroups your data into a specified number of bins (it acts like a form of filter on your data). The output of Histogram is a new Field with connection-dependent data. The connections are the bars on the histogram (which can be plotted). The height of each histogram bar is proportional to the number of samples of original data that occur in the range covered by that bar. You can feed the output of Histogram through AutoColor then Plot to get a colored plot of the data distribution.

If the aspect ratio of the Plot is distorted, you can correct it in the Plot module. This will stretch the Plot out in either the X or the Y direction until you achieve the look you want. Visual designers recommend an aspect ratio of approximately 4 units wide to 3 units high; since this is also the aspect ratio of television, your image will be ready both for video and for print.

Be aware that “binning” your data with Histogram can sometimes create rather arbitrary distributions. It is important to make this clear to the viewer of your visualization. For example, by carefully selecting bin size, you may turn a unimodal distribution into a bimodal one. Which distribution is correct for the phenomenon under study must be determined by the underlying science, not by the arbitrary picture you create.

On the other hand, if you wish to actually redistribute your data rather than just show a histogram of its distribution, you can use the Equalize module. The output of this module is essentially the same scalar Field you fed into it, but the data values have been changed to fit the specified distribution. By default, the data values are changed to approximate a uniform distribution, but you can create your own custom distribution, like a normal Gaussian curve. Equalize is useful to reduce extreme values back to a range similar to the majority of data values. You may also wish to experiment with other data “compression” and “expansion” techniques by connecting your data Field to Compute and applying a function like “ln(a)” or “a^2”, where “a” is the input Field.

Rubbersheet

Another technique used to visualize data collected on a 2-dimensional grid is sometimes called a “height map.” In Data Explorer, the Rubbersheet module will generate this for you. Conceptually, a height map is drawn by elevating the 2-D grid into the third dimension. Call it the Z dimension, with our original grid lying in the X-Y plane. The height or Z-value given to each vertex of the original grid is proportional to the specified scalar data value at that vertex. If the data were vector data, you could elevate the grid by the magnitude of the vector, since magnitude is a scalar value. The result usually resembles something akin to a relief map of the surface of the Earth with hills and valleys.

However, this brings up an important point that will occur elsewhere in Data Explorer (and visualization in general). Remember that the original data were collected on the X-Y plane (for example, our grass-counting botanist’s data). It is one thing to indicate the different distributions of grass species by showing a 3-D plot of the numbers using a height map. But it is not correct to say, then, that the data values so shown were collected from these 3-dimensional positions: that would imply the botanist counted grass species growing in mid-air! This might be true in the Amazon, but not in Kansas.

That is, we may have counted 2 species at the grid point [x=0, y=0]. If we Rubbersheet using the species count as the Z deflection value, our 3-D height map will now have a point at [x=0,y=0, z=2] (if the Rubbersheet “scale” is 1.0 and the minimum count in our data set is 0). The data was not collected at that point but rather at [x=0,y=0, (z=0)]. For our convenience, Data Explorer maintains the original data values as if they were attached to the original grid. It is your responsibility to remember and, if necessary, make it clear to other viewers that the representation of the data in 3-D is not a “realistic” image of the original 2-D sampling space. Rather, Rubbersheet is used to visualize the “ups and downs” in the data Field as actual differences in height. This is a very powerful visualization technique because of our familiarity with actual heights in everyday experience. One simple way to show viewers the difference is to make two copies of the Field by taking two wires from the output tab of the Import module you use to import the data Field. Connect one wire to a Color module with a Colormap attached, but leave the Field 2-dimensional. Arrange the 2-D colored grid such that the viewer is looking straight down on it. Connect the second wire from Import to Rubbersheet and then use a second Color module, but run wires from the same Colormap as you used to color the first copy. The second copy, a 3-D colored “height Field”, can then be rotated into a “perspective” view. The result will be a Field both colorized according to the data values and also elevated into the third dimension according the same data values. This redundancy is often more instructive than either visualization technique used alone.

Transformations and Structuring

Rotate, Scale, Translate, and Transform are all special types of operations that change the location, orientation, or size of objects in your scene. These operations can be performed anywhere in a visual program. You can create “hierarchical” motion by attaching Rotates and Translates to individual objects, then Collect these objects together and attach another Rotate and Translate to the Group (output of Collect). In this fashion, you can individually rotate members of the group independently of each other, or you can rotate the entire group as one.

By default, many modules operate on the “data” component. We have been treating “data” as a special kind of numeric Array, separate from “positions” and “connections”. We mentioned earlier that you can have several different “data” components, but each must have a unique name; for example, your input data file can contain “positions”, “connections”, “temperature” (data), and “wind” (data). For this example, assume that “wind” is a 3-D vector.

Using the Structuring category tools Mark and Unmark, you can convert any Field component into the “data” component. When you Mark “wind” for example, the old “data” component (if any) is moved into a safe place called “saved data” and the “wind” values are copied into the “data” component. Since “wind” is a 3-D vector, the new (current) data component becomes a 3-D vector also. The Compute module is used to make changes in the data component of a Field. So by multiplying the first (x) component of our 3-D “data” we are, in effect, scaling in X. For example, the Compute expression in this case would be “[a.x * 2.0, a.y, a.z]” to double the size of each x component of each “data” point while leaving the y and z components the same. Any module connected to the output of Compute will see the scaled “wind” values as the “data” component of the Field. However, the old unscaled “wind” values are still kept in memory, also. By connecting other modules to the originally imported “wind” values, you still have access to those original values, at the same time. To operate on the “temperature” data, first use Unmark to return the “data” to the “wind” component. The result will be to place the scaled “wind” values into the “wind” component for all modules connected to the output of Unmark. Unmark also copies back all values from “saved data” into the “data” component. Then, you can Mark “temperature” as “data” and perform operations on it, if you like.

Since “positions” are also 2-D or 3-D vectors, you can Mark “positions”, perform operations on the grid itself, then Unmark “positions” to perform operations on the “data”. With a little knowledge of the correct matrix operations, it is possible to simulate the effects of rotations, translations, and scalings using this Mark technique. You can warp flat grids into cylinders or polar coordinate systems or create more complex objects like cones. In fact, there are already many macros available in the Data Explorer Repository that handle these types of operations using this technique, which you may wish to download and use yourself. (For the Data Explorer Repository, see “Other sources of information” on page xxiv.)

Vector Fields

Vector-valued data sets occur very frequently in visualization. Data Explorer offers three ways to visualize vector Fields: vector glyphs, streamlines, and streaklines. For this example, assume that we acquired data on wind velocity and direction in the atmosphere.

Recall that a “glyph” is a visual object; a Field of glyphs is made by copying a generic object, positioning each copy appropriately, and scaling or coloring each copy according to the data associated with that sample point. Vector glyphs resemble arrows or rockets and are generated for you by the Glyph or AutoGlyph modules. A vector Field, like any Field, must have a positions component to identify where the vector-valued data was sampled (even if the data is connection-dependent, it still requires positions). For Glyph realizations, a “connections” component is not required, but it may exist if the Field contained it for other purposes. Of course, a data component containing a vector quantity is needed. Each vector glyph will point in the direction of the vector given by the datum at that point, with the base of the vector fixed at the vertex position (sample point) for position-dependent data. The base of the vector is located at the center of the connection element for connection-dependent data. The length of each vector glyph is scaled based on the vector “magnitude”, relative to all the other vectors in the data Field. Glyph and AutoGlyph offer a number of modifications you can make to achieve the appearance you desire. The effect of glyphing a vector Field is to create a “porcupine” plot with lots of arrows sticking out in various directions. This can become hard to interpret if there are many vector data points or, if one area of your data has very large values, the vectors may intersect or occlude each other. You can use the Reduce module (in the Import and Export category) to downsample the original data Field and thereby decrease the number of vectors in the image. Picking a reasonable reduction factor will permit the viewer to see the overall vector Field direction(s) while reducing the visual clutter.

You can also use the Sample module to extract a subset of points of the data Field. For example, you can select a subset of points lying on an isosurface; these data points can then be fed to Glyph. The effect in this case is to show the vector Field direction and magnitude sampled at the surface of constant value. This is another technique to reduce the number of vectors glyphed at the same time and may make it easier to perceive the structure of the vector Field.

Another technique for visualizing a vector Field relies on the concept that there exists a potential flow direction through the Field. Imagine releasing some very light styrofoam balls into our wind Field; each ball has a streamer attached to it. (Gravity and friction are ignored by the visualization tool; of course, you may have accounted for these forces in the simulation that modeled the vector Field, if these forces are relevant to your science.) We release the balls at one instant on one side of our Field and after they have passed through the Field, we take a snapshot of the streamers. This type of image is essentially what you get with the Streamline module. Streamline is used to visualize a flow Field at an instant in time; it assumes that you have a particular measure of a vector Field and wish to study the "shape" of that static Field.

Streamline produces a set of lines that show the flight path of each "ball and streamer." You can indicate the starting positions of these paths in a number of ways: essentially, any kind of object with positions can be the designated start point or points for Streamline. For example, you can use the Sample module to extract an arbitrary subset of positions from an isosurface, then treat this subset of positions as valid starting points for Streamline. You would see a set of streamlines that began *on* an isosurface and then traversed your vector Field. If you want to visualize the streamers' associated "twist," use the Ribbon module and use the curl and flag parameters of Streamline to force computation of the vorticity field. Streamlines can also start from a Grid, a list of positions, or a Probe. The Probe is a handy way to interactively investigate a vector Field; Probe tools are selected from the Special category. They are manipulated in the Image window; select **View Control...** from the Image window's **Options** menu, then choose **Cursors** from the **Mode** pop-up menu. Any Probes that you have placed in your visual program will be listed in another pop-up menu, so you can pick the one you wish to interactively manipulate. By dragging the probe through the vector Field, the Streamline starting point will follow the mouse pointer (again use **Execute on Change** to see this happen interactively).

Streakline is used to study a dynamic vector Field. Streakline is equivalent to taking a series of snapshots as our styrofoam balls and streamers (or just the balls without streamers if you like) fly through the vector Field, but with the additional fact that each time we take a snapshot, we import the next time step of our Field. That is, at each moment, we provide new data for vector direction and intensity at each sample point. As a result, you would expect the direction and speed of the balls and streamers to change as their flight is affected by the changing Field. This technique is often referred to as "particle advection."

Note that both Streamline and Streakline perform interpolation, so both modules require that your input vector Field has positions, data, *and* a "connections" component.

Volume Rendering

Another way to examine data collected throughout a volume of space is called *volume rendering*. Imagine a glass bowl full of lemon gelatin. Holding it up to a light, you can see through the gelatin because it is somewhat translucent. Now imagine that you have added strawberries to the bowl of gelatin before it set up. You can see the strawberries embedded in the gelatin. What is really happening, visually? Light shines through the mass of gelatin "accumulating" color. If you look through the top corner, it will appear somewhat less yellow than if you look through the thickest part. If the light strikes a strawberry as it passes through the gelatin, your eyes will detect an orange object with a distinct outline, which of course enables us to find the location of the strawberries in the volume of gelatin. The strawberry appears orange because its red color is partly occluded by the yellow gelatin: nevertheless, our brains convert the strawberry color back to red because it is a familiar object. If someone has added a fruit unfamiliar to you, you will have a hard time identifying the true color of the fruit, since our brains are not good at performing subtractive color calculations.

Volume rendering a data space yields an image something like our bowl of gelatin. By default, a volume rendering appears somewhat transparent. As light passes through from behind the volume toward your eye, it is absorbed more in areas of densely concentrated values. These areas will appear to be more "opaque." If you color-map your volume according to the data component, you will see indistinct colored areas in their relation to each other. For more detail on the "dense emitter" model used by Data Explorer, see "Opacities Component" on page 23.

If we are looking for those areas of rain formation within a rain cloud data volume, we do not have a built-in conception of the "correct" color for such an area. The colors assigned will come from the color map we construct. If we map the 12 degree C area to red, as in the example above, the red-colored rain-forming areas seen through a yellow cloud will, in fact, be perceived as orange areas. We can temporarily hide the yellow cloud (by changing its opacity to 0.0 and its color to black) and entrain ourselves to see the red regions by themselves.

This is a fine point of perception, but it is important to be aware of. Perception of natural objects is greatly modified by psychological memories and judgements about their "correctness" in size, color, mass, and relationship to each other. Once we move into the abstract world of visualization, we have no firm psychological constructs on which to base our perceptions. While this may imply that we are working with a "clean slate"—no preconceptions, and an unbiased scientific viewpoint—just the opposite happens: we seek to impose interpretation on the scene and may ascribe invalid attributes to objects as we try to derive "meaning" from the scene. On one hand, this is precisely why we imaged the volume in the first place! We want to derive patterns or shape and then figure out why they exist. On the other hand, we can be fooled by our own eyes if we are not very careful to comprehend and explain to others exactly the assumptions we make as we convert our sample numbers into colored images.

By the way, you won't find a specific module named VolumeRendering. As it happens, any volumetric Field can be directly rendered by the Image module or the Render or Display modules. So if you simply Import your volumetric data, run it through AutoColor, and attach it to Image, you will get a colored volume rendering of your data space.

A.3 Design for Interactive Use

Data Explorer is first and foremost designed for interactive exploration using data as the raw material for creating, modifying, and understanding imagery. As such, the system is designed to permit operation at several different levels of expertise: “exploring”, “authoring”, and “programming.” You can engage in exploring by opening previously created visual programs (e.g., the visual programs provided with Data Explorer) and changing values of the Interactors found in Control Panels, changing values in the Image window controls, or by manipulating the Sequencer (if one is provided). You author a visual program when you place and interconnect additional modules on the VPE or reorganize connections between modules already present, or when you create a new visual program from scratch. At the advanced level, you can learn how to write and add your own custom modules to Data Explorer. This involves writing in a traditional programming language like C, taking advantage of a rich function library provided, compiling your new module, and running your customized version of Data Explorer. While it is nice to know you can do this, do not worry if you are not a programmer, because you may never find a need to write your own module considering how powerful the “stock” Data Explorer tool set is.

The next section addresses those who want to create new visual programs, So here are a few tips on design for interactive use.

Interactors and Control Panels

Interactors are special two-part modules. To incorporate an interactor into your visual program, you select the preferred type of interactor from the Interactor category, place it on the VPE *canvas* just like any other module, and connect it to the appropriate input tab or tabs on other modules. The interactor module that appears on the canvas is called a “stand-in.” To use the interactor interactively, you also place an instance of the interactor in a Control Panel window. The interactor in the control panel represents the actual manipulator used by the user exploring the data. When a value is set by a user, it becomes the new output of the interactor stand-in and is thereby fed to the modules connected to the stand-in’s output. Interactors have different appearances depending on its type; numeric (integer or scalar) interactors can be made to look like dials or sliders, while string interactors give you a place to type in a string. List interactors let you keep a list of items: there are lists of strings, vectors, values, integers, and scalars.

It is good “programming” practice to set interactor minima, maxima, and increments to reasonable values. For example, a Scale module will accept a value of 0.0, but the effect will be to make the scaled object disappear! That is usually not desirable; set the minimum permitted scale value to be a positive value greater than 0.0 if you do not want to confuse users of your visual program.

Some interactors can be used in a different mode than interactive: these are called *data-driven interactors*. Scalar, Integer, and Vector interactors, (and their respective List types) all have input tabs of their own. By default, all the numeric interactors have arbitrary ranges preset to –1,000,000 to +1,000,000. Clearly, these will rarely be the appropriate ranges for your data. As part of good interactive visual program design, you, the visual program author, would like to restrict these ranges to the “correct” values for the input data sets. But if you are building a visual program for use by others, you won’t know in advance the ranges of data sets the user will import. If you build your visual program such that the data

Field (the output of Import, for example) connects to the input tab on one of these interactors, the correct maximum and minimum values will be automatically set the first time the visual program is executed, and they will be updated appropriately as the input data set changes. Thereafter, the user cannot accidentally exceed the range of values by turning a dial or sliding a slider too far in one direction or the other.

Data-driven interactors can be directly driven by Compute functions that might in turn be connected to a Sequencer or other data Fields or data components in a Field. As just a simple example, a Sequencer could emit a series of integers from 0 to 360; a Compute can turn the integers into floating-point angles in the range from 0.0 to 1.0 then make this new number the first component of a 3-vector ("[a/360.0, 1.0, 1.0]"); then Convert can change this HSV vector into RGB. Connect the output of Convert to a Vector Interactor and feed the interactor output to Color. The result is that the Sequencer will make the color of an object attached to this Color module pass through the entire spectrum of hues; simultaneously, you can watch the RGB values change on the Vector interactor in the Control Panel.

You can have as many Control Panels associated with a visual program as you like. Furthermore, a handy feature is that the same "stand-in" (the Interactor module that appears in the visual program) can have multiple interactor instances associated with it. This means you can have both simple Control Panels and elaborate Control Panels with commonly needed interactors appearing in both. When you do set things up like this, you will notice that the multiple instances of the interactors will always maintain the same value: as you change the value in one Control Panel(s), the associated interactor(s) in the other Control Panel will stay in perfect agreement.

Control Panels can be named and accessed by name. This allows you to set up hierarchies or even rings of Control Panels. You might choose to make a simple panel with only the most commonly used interactors, then create additional panels with less-used interactors. The main panel can then be set up to access the subpanels by name, using the Control Panel's **Panels** menu. Select **Open Control Panel by Name** to see the list of other Control Panels accessible by the current panel.

It is very important to create sensible labels for the interactors in your Control Panels. Data Explorer will automatically assign a name to a new interactor that reflects the name of the module and input to which you have attached the interactor stand-in in the visual program. However, this name tends to be too generic, especially if you have several interactors connected to several similar modules. For example, you connect a Scalar Interactor to the "value" input of an Isosurface module. The interactor label in your control panel will acquire the title **Isosurface value**. But if you also place another Scalar connected to a different Isosurface, you will end up with two interactors with identical names. So it is incumbent upon you, the visual program author, to change the names of your interactors to reflect their function in your visual program. See 7.1, "Using Control Panels and Interactors" on page 128 and "Using Interactors" on page 142 for the instructions on using Control Panels and Interactors.

A very handy interactor is the Selector. This interactor lets you construct a pop-up menu containing one or more string items each associated with a value, either a scalar (including integer scalar), vector or a string. Selector has two outputs, the value and the string you have entered. This allows you to present a menu that describes clearly what choices the user has, and when the user picks an item, Selector outputs both the value (through its left-hand output tab) and the string choice (through the right-hand output tab). The right-hand output can be attached to modules that accept string input. Remember Format? You can use this technique to change a caption depending on the user’s current Selector choice.

If you use integers as the “values” in your Selector, the left-hand output will be the currently selected integer. You can direct this numeric output of Selector to any module that takes an integer; a common use is to connect the integer output of Selector to a Switch module.

The Switch module (in the Structuring category) uses a number to pick which of several inputs to pass to its output. Suppose you have connected an Isosurface to the first “input” (the second tab) of Switch and a ShowConnections to the second “input” (third tab). You have also constructed a Selector menu to offer the three choices: 0 = Both off; 1 = Show Isosurface; and 2 = Show Connections. The left-hand output of Selector is connected to the first tab (the “selector”) of Switch. Now when the user chooses item 1 on the menu in the Selector interactor (located on a control panel, of course), the number 1 is emitted and received by Switch. Switch then lets the isosurface pass through. One more trick: you may want to allow both the Isosurface and the ShowConnections images to appear at the same time. Use a Collect module just before the Switch. Attach the Isosurface output both to Switch input 1, and to Collect; similarly, attach the ShowConnections output to Switch input 2 and to the Collect. Now add a fourth choice to the Selector menu: 3 = Show both. Attach the output of Collect to “input” 3 of Switch, and you have provided this new capability to the user. By the way, the value “0” will always turn off all output from the Switch; you do not need to provide a “0” valued choice if that is not appropriate; in other words, if you always want Switch to pass at least one item.

The default settings for Selector are 0 = off, 1 = on. You may find this handy as you begin to develop more complicated visual programs containing a number of objects in the Image window. As you develop each “subnet” (that is, a branch of the visual program that yields a particular visual object), attach it to a Switch and add an on-off Selector. Change the label of the Selector interactor in the Control Panel to identify the object it controls. Run the output of each Switch to a Collect (add inputs to Collect as needed), then to Image. This way, you have a whole panel of Switches, allowing you to turn off and on each object in the scene. This will decrease the amount of time you wait for all objects to be rendered if you know that certain ones are OK but wish to test new ones in the scene. This technique is easier than connecting and breaking wires, too.

Transmitters and Receivers

In the Special category, there are two modules, Transmitter and Receiver, that should be used in larger visual programs. Each Transmitter can “broadcast” to any number of identically named Receivers. The name you choose for the Transmitter is analogous to a radio station’s broadcast frequency. Receivers with the identical name are like radios tuned to that channel. Like radios, more than one Receiver can receive from a single Transmitter; more than one Transmitter can broadcast, each on a different frequency, requiring differently named (“tuned”) Receivers. This means you reduce the clutter of wires looping all over the screen in the VPE. But the real advantage of Transmitters and Receivers is that you, the visual program author, can provide meaningful names that then appear on the modules in the visual program. This is a handy way to provide some visual documentation of the way the visual program is wired.

Although you can add Transmitters and Receivers to your net at any time, and do not have to add them in pairs, you will find it is easier to add one or more Receivers to a net right after you place and name the corresponding Transmitter because Data Explorer automatically gives Receivers the same name as the most recently placed Transmitter. However, if you decide to add a Receiver later, just be sure to double-click the Receiver module and set its name to the name of the Transmitter you wish it to receive from. Changing the name of any Transmitter will automatically change the names of all associated Receivers, but changing the name of a Receiver affects only that specific module.

A good way to use Transmitters is to broadcast "global variables", to use the terminology of traditional programming. For example, you are allowed only one Sequencer per visual program, but, as discussed earlier, the output of Sequencer may be used by many "subnets" to perform various functions. You may find it most convenient to place the Sequencer connected to a Transmitter you name "sequencer" near the top of your visual program (the lowercase "s" helps remind you of the function of this Transmitter, but you may use any name you like). Then, wherever in the net that you need to receive the current value of the Sequencer, attach a Receiver named "sequencer."

Another global that you may want available is the path name of your current work directory. Attach a String Interactor to a Transmitter. Then pick up this "channel" with Receivers throughout the visual program, for instance, as an input to a Format module (the Format template must include a "%s" as a place-holder for a string input). You will find this especially convenient when you give your visual program to a colleague who will naturally place the visual program and data files in a differently named subdirectory on your colleague's workstation. By simply changing the name in the Interactor to identify the name of the work directory on the new machine, the visual program will be back in business. If you had "hard-coded" the name of the path into several modules on your visual program, the new user would have to hunt down all these references and do a lot of extra typing.

Documentation

When you author a visual program or create a macro (a special kind of visual program, discussed in "Creating Macros" on page 149), you should describe its function using the Comment capability (found on the **Edit** menu in the VPE). This Comment will be viewable by other users (or yourself) when they run your visual program and choose the Help menu item **Application Comment**.

Another brief kind of documentation is available in each module. Double-click a module, and its default name is shown in the "Notation" box at the top of the module description dialog box. You can add to or change this notation. This is particularly helpful in Compute modules to describe the meaning of the expression, for example, "square root of pressure per cubic inch." And any time you do something "tricky" with a module, enter a note to yourself in the Notation box for later reference.

Compute also offers a useful way to document the terms of an expression. Each input to Compute can be given a meaningful name (the default names are simply a and b). If you like, you can change the input letter names to words, like "pressure" and "scale", then use an expression like `sqrt(pressure / scale) * 0.5`. You may add more inputs to Compute by simply pressing **Ctrl+A** when you have selected the Compute module.

You can also add annotation text directly to the canvas using the **Add Annotation** option of the Edit menu of the VPE (see “Adding Annotation to a Visual Program” on page 115). And you can segment your visual program with pages (see “Creating pages in the VPE” on page 115).

A.4 Design for Video Output

Video production is only one useful output from a Data Explorer visual program. Videotape has the advantage of portability: you can send a video to almost anyone these days because of the proliferation of consumer VCRs. But video has the great disadvantage that you can no longer make interactive changes in the images: you cannot “explore” any more once you have committed your images to videotape.

Unfortunately, high-definition digital television is not widely available yet. It is very important to be aware of the technical limitations of standard analog television, especially as it differs from workstation monitors.

The two biggest problems encountered in moving images created on a high-resolution workstation to a consumer television (monitor or VCR) are the loss of resolution and the inability of consumer TV to accurately render color. (These remarks are basically true for both the American NTSC TV system and the European PAL system. This is not meant to be a technical description of either system.)

TV Line Resolution

Resolution losses are most evident if you use single-width lines. Workstations have both a higher number of vertical and horizontal lines on the screen, and a much higher “refresh” rate than consumer TV. However, on TV, the alternating lines (odd numbered, even numbered) are “refreshed” or painted on the monitor at slightly different times. As long as the scene contains objects that span more than one of these lines, our eye-brain system is fooled into believing that the entire object is always present, due to the phenomenon of “persistence of vision.” But when you use single-line width horizontal lines, the lines will visibly flash, clearly showing that they are being drawn only half of the time.

Related to this problem is the condition in which you rotate a grid of single-width lines slightly away from horizontal. This will generate an optical effect called a “moire pattern”, in which curved lines appear where none are actually present, and, this frequently causes colors to appear that are not in the original signal. Both of these effects can be very distracting.

Finally, single-width vertical lines will not have the same color! Because of the way consumer TV color phosphors are aligned, a vertical line at one location may be blue, but if you move it slightly it will become red. A grid with single-width vertical lines will appear to change color as you translate the grid in a horizontal direction.

So what is the solution to all these problems? Do not use single-width lines, ever. Data Explorer's Tube module is the easiest way to fix most of these problems. Tube generates cylinders around any kind of field with line connections. If you have created a mesh of lines with ShowConnections, for example, you can run this visual object through Tube to “fatten” up the lines. Tube permits you to choose a diameter that looks right. As long as you make the tubed lines bigger than one TV line width, you will have solved the problem.

Be aware that single-width line text or captions will become virtually illegible on TV. To get better-looking results using the “stroke” fonts (originally designed for plotters), you can use Tube. Another technique is to use a multiple-line font, such as the “roman_d” font supplied with Data Explorer. The best solution is to use an “area” font that is made up of characters containing polygonal faces rather than single lines. Data Explorer provides a font called “area”, or “pitman”, which uses polygonal faces.

Another tip about text is that due to the much lower resolution of TV, you must be careful to keep text large! Ideally, use a size that permits only about 30 to 40 characters to fit across the width of the screen. Fine detailed text annotations may look good on the workstation, but will become blurry little globs on TV, defeating the whole purpose of annotating your video for your viewers. Try making some text in different sizes, then dub to VHS videotape. Can you still read the text? If so, the size is probably sufficient for general use. If it is too mushy to read, increase the size. For best legibility use white or yellow colored text.

TV Color Resolution

Standard TV is simply not capable of correctly rendering fully saturated colors, like red (in particular) or blue. Large areas of fully saturated colors will pulse and “bleed”; that is, they will smear to the right (due to the direction the TV raster scan is moving). This smears any sharp edges on your objects and will severely degrade the quality of your visualization. The color problem can best be dealt with by never using fully saturated colors. Instead, when building your color maps, lower the entire Saturation curve to about 0.8. Although this will look much more pastel than you might prefer, once you have converted the images to TV, these colors will brighten up again. What looks kind of pink on the RGB workstation monitor will usually be much redder on TV. Of course, if you are producing images for another medium, like a color printer, you can set the color saturations appropriately (fully saturated may be correct in that case: the tips in this section are to help you make better video recordings).

Animation and Frame Rates

In making animation for TV, you must be aware of “frame rate.” Just like all the other references to sampling in the discussion above, TV “samples” time at 30 frames per second (25 in PAL). That means that at most you can have 30 time-step changes per second of video (unless you choose to skip some time steps in your data). Generally, you may want to show many fewer changes than that or the phenomenon may go by too quickly for the viewer to comprehend. On the other hand, when you rotate an object you want to make as many small changes as you can afford. The result will be smooth animation rather than jerky cartoon-like movement. One rule of thumb is to rotate no faster than 3 degrees per frame. That means that your object would rotate 90 degrees in one second, or 360 degrees in 4 seconds. Like any rule of thumb, this can be adjusted depending on the case at hand. For example, it is often useful to record the rotation at more than one speed. The human visual system will detect different levels of detail in an object depending on its motion rate. This can be used to your advantage, to get double-duty out of your visualization. Record and play it at one rate, and viewers will see one aspect of your data; play it faster or slower, and different details will be noticed. And often, it is good practice to let an animation "loop" a few times, allowing the viewer to observe the entire process from beginning to end.

The Sequencer also can generate "palindromic" motion in which the object swings back and forth rather than jumping from the end of a series back to the beginning. Be sure that you use this feature in a meaningful way: time steps shown in reverse order imply time running backward. Annotation is definitely required in this case!

A.5 Presentation: Issues and Techniques

Visualization is used to represent natural phenomena that are inherently visual themselves, but probably more often, it is used to "visualize" non-visual phenomena. The process of making something visual means making choices on the part of the program author or designer. Clearly, without a sound scientific basis for these choices, this can become a purely artistic venture. While computer graphics can be used to make beautiful artwork, that is presumably not the point of using visualization to help study, analyze, or understand data. This does not mean that you should forego good design in making your visualization scene understandable. Remember and use the "rules" of design mentioned above, including proper, legible annotation, reasonable choices for colors, and so on. These things are determined partly by the medium you are working in and partly by the rules of good layout and design.

But what color is a magnetic field? What color is hot? What color is high? How fast should molecules vibrate? How quickly should a metallic surface move as it changes phase? These decisions must be made by the program author. Probably the three most critical choices are color, scale, and speed.

In visualization, color is used precisely because it is *not* realistic. That is, to emphasize an area of interest, red is commonly used. Or, a strong contrast color can be used against a field of fairly neutral colors. However, there are some cultural color choices that you may find inappropriate to violate. For historical and to some degree natural reasons, we tend to make color gamuts that indicate red as the "highest" and blue the "lowest." Particularly with temperature, we can associate blue with "cool" or water/ice color, and red with "hot" or flame/sun color. To some degree, this gamut is related to the color of heated metal, but of course, the metal color does not pass through green at the midway point, and the color scale does not end at white like white-hot metal, so this too is only a loose analogy. But try inverting a color map of temperature to make red cool and blue hot and you will probably find you have to perform mental gymnastics to interpret it "correctly." If you are mapping altitude, however, red is not necessarily best associated with the "high" point: after all, the highest altitudes are snow-covered and lower altitude deserts are frequently "red-hot"! Actually, color-mapping altitude is almost purely an artistic endeavor, but at least it has a long history and literature in cartography. Consulting the "traditional" textbooks for a field may indicate how users in that discipline "prefer" things to be mapped. It is generally unwise to start a new schema for your visualization if you wish it to be immediately accessible to other viewers familiar with the discipline. But relating new ways of visualizing data to the old methods may be a good way to provide new insights for everyone involved.

Remember that to use interpolation, the basis of your assumptions is that the phenomenological space studied is continuous and linear. If you have reason to believe the sampling was not done over a domain that can be linearly interpolated, you should certainly not be using linear interpolated images to understand the data.

You may need to collect more data on a finer grid to resolve such problems. Since Data Explorer supports irregular grids, this is not a problem for the software, as long as you provide the correct data sampling. Also, be aware that trying to read too much detail out of an image is an error. You cannot accurately assess detail at a resolution equal to or less than your sampling rate (the Nyquist law states that you cannot derive valid signal from noisy information at less than twice your sample rate). For example, occasionally, you will see peculiar color artifacts that arise when data and therefore interpolated colors change rapidly at the scale of the sampling mesh. In those cases, the best bet is to “zoom out” to see only the big picture: do not try to read between the lines!

Related to sampling rate in space is sampling in time. Be sure you have collected enough time step detail to ensure you have not completely missed some important transitional state that might have occurred in the middle of an animated sequence. It is acceptable to skip through the entire range of time steps during the development of your animation, but be sure to fill in the gaps before the final presentation is analyzed.

As in traditional statistical plotting, a computer can all too easily permit the author to scale objects or graphs into wildly distorted aspects. In charting, there are some simple rules of thumb: it is often suggested that the aspect ratio (height/width) be about 0.75 to 1.00 for a 2-dimensional chart. This may require rescaling one axis, and naturally, both axes and their scales must be shown. It is also bad form to start an axis at one point then create a break part way along, causing a visual foreshortening. And it is also inappropriate to start an axis at a point other than the origin if the intent of the chart is to represent absolute amounts of quantities being compared side by side. All of these rules of thumb are employed to make “good” charts; nevertheless, these rules are too often violated even in the mainstream media.

Unfortunately, these traditional rules of scale do not help us much when we create 3-dimensional objects of arbitrary shape. So it becomes incumbent upon you to make sensible decisions in depicting objects never before seen by any viewer. It will be very easy to exaggerate a 3-D height field by changing the scale factor in Rubbersheet. You can make the one high point in the data leap as high as Mt. Everest. If that point is in fact a special value in your data, this may be an appropriate thing to do. If not, you may wish to choose a scale better suited to depicting the entire surface. On the other hand, if there are peaks, you must avoid “crushing” the entire surface to lessen the high points. Doing so could lead to potential misinterpretation of your results.

For many researchers, Data Explorer will be the first program they have used that permits them to create and view animation or motion playback of their data. This new temporal dimension is often a source of problems until the author gets the hang of things. Here are a few tips as you develop your own “moving pictures.”

First, remember that your viewers have never seen this phenomenon before. Give them a chance to absorb it: looping the entire sequence is usually helpful. You do not want to bore the viewer to death, but visualization is not a TV commercial: cutting to a new scene every two seconds is not a good editing technique for communicating difficult visual information. As we discussed in the section on Animation, showing the same sequence at more than one speed helps a viewer notice different information in the very same scene.

Visualization allows users (fortunately) to wildly distort time scales. One video may show the movement of tectonic plates, another the gyrations of atoms in a gas. One scale is millions of years, the other billionths of seconds, but both are brought into the “video” scale of one frame every thirtieth of a second. Clearly, you must use some kind of clock annotation, especially if you plan to change playback rates, and even more importantly, if you plan to show different data sets using the same type of animation. The user must be given a proper sense of how two animations compare in their duration if sense is to be made of these animated sequences.

However, humans are not particularly good at visual comparison from memory. We are good at pattern recognition and comparison, but we have inadequate temporal rate memories; we do not remember detail in relation to time because we do not have good time-keeping reference systems in our brain. That implies that you must either choose to show comparisons based on precisely the same time duration and playback rate (thus factoring out the time dimension), or, much better, show two motion sequences at the same time in the same picture. One way to accomplish this is to render two sets of images, then use the Arrange module to construct an animation showing the two sequences side by side. This technique is important if the two phenomena vary in a scientifically critical way during the process; for example, if one phase change event is virtually complete after 40% of the entire time step series and another phase change after 60%, this may represent one of the important findings of your research. But if you show the viewer first one sequence, then the other, very few people will be able to make a solid visual comparison from their memory. It is much more visually impressive to show the two phase change simulations side by side, starting at the same time, and proceeding for the same number of time steps.

Animation must also proceed quickly enough for the mind’s eye to perceive it as animation. Imagine taking each time step of your simulation, making a 35mm slide, and loading up a slide carousel with ninety slides. A viewer who is shown each slide for 5 seconds is unlikely to perceive the “motion.” Put on videotape, the same sequence of images takes only 3 seconds. This may be too fast: the entire event may flash by too fast for the viewer to see any change. You may need to double-record each image (i.e., slowing things down by one-half) making the video take 6 seconds. Another way (more computationally expensive) is to generate twice as many raw data files and twice as many images. This will yield smoother animation, but may be too costly for your resources. Of course, some events can be shown in 3 seconds: maybe everything stays the same for 1.5 seconds, then “pops” into a new configuration. Slowing this down too much might hide the importance of the sudden transition to a new state. Again, you, the user familiar with the field and with the phenomenon become a judge and a designer. You have to make wise decisions based on a desire to accurately and honestly depict the behavior under study with the purpose of illuminating other viewers, not impressing them with spectacular computer graphics displays.

Appendix B. Importing Data: File Formats

Importing your data into Data Explorer will be the first step in creating a visualization of that data. In order to take this step you should have some understanding of the Data Explorer data model and a working knowledge of a Field. An informal description of a Field is provided in Chapter 2, “Introduction to Visualization” on page 7. A formal description is given in Chapter 3, “Understanding the Data Model” on page 15.

A number of methods for importing data are available for use with Data Explorer: the General Array Importer, Data Explorer native file format, netCDF, CDF, and HDF.

B.1 General Array Importer: Keyword Information from Data Files

In addition to the syntaxes for the grid, points, and positions keywords described in *IBM Visualization Data Explorer QuickStart Guide*, it is also possible to derive information for these keywords directly from the data file. This allows you to write “filters” for specific applications that output their data in a set format which includes the grid size within the file.

The syntax for the grid keyword is (in addition to the syntax given in 5.3, “Header File Syntax: Keyword Statements” on page 85 in *IBM Visualization Data Explorer QuickStart Guide*):

```
                        bytes n                                      bytes n
grid =  [format],[type]{lines n   , [skip, width] [skip, width],..  {lines n, ...
                        marker string                                marker string
```

where:

format
: is the format in which the grid values will be found, and must be one of the following: **binary**, **ieee**, **text**, or **ascii**. The first two parameters are synonymous, as are the second two.

type
: is the type of the values, and should be one of the following:

byte	**int**	**short**
unsigned byte	**signed int**	**signed short**
signed byte	**unsigned int**	**unsigned short**

Note that in each of the three groupings shown here, the first and second (reading down) are equivalent to each other.

bytes, lines, and marker
: specify where to begin reading the grid values.

skip, and width
: are optional and should be used when two pieces of information are on the same line with other information separating them (see Example 2). If necessary, different portions of the grid specification can be read separately by repeating the bytes, lines, or marker specification (see Example 3).

Example 1

Suppose that the data file contains the following first line:

```
dimensions 100 300
```

You can specify that this information is to be derived from the data file by the following statement:

```
grid = lines 0, 11, 3, 1, 3
```

This specifies that 0 lines are to be skipped. Then 11 characters are skipped, and the first grid dimension is read from 3 characters. Then 1 character is skipped, and the second grid dimension is read from 3 characters.

You could also have used the statement

```
grid = bytes 11
```

which simply specifies that the grid information will be found after skipping 11 bytes in the file.

Example 2

Suppose the that data file contains the following line (not at the top of the file)

```
xdim = 5 ydim = 20
```

You could use the statement

```
grid = marker "xdim =", 0, 2, 8, 2
```

This specifies that one should start reading after "xdim =," read the first dimension from 2 characters, skip 8 characters, then read the second dimension from 2 characters.

Example 3

suppose that the data file contains the following lines

```
xsize = 20
ysize = 30
```

You could use the statements

```
grid = marker "xsize =" X marker "ysize ="
```

or

```
grid = marker "xsize =",0,3 X lines 1,8,2
```

The first specifies that the first dimension should be read following the marker "xsize =," and the second dimension should be read following the marker "ysize =." The second statement specifies that the first dimension should be read from 3 characters, after skipping 0 characters following "xsize =," and that the second dimension should be read from 2 characters after skipping 1 line and 8 characters.

The syntax for the **points** keyword is (in addition to the syntax given in 5.3, "Header File Syntax: Keyword Statements" on page 85 in *IBM Visualization Data Explorer QuickStart Guide*):

```
                          bytes n
points =  [format],[type]{ lines n   , [skip, width]
                          marker string
```

where:

format is the format in which the grid values will be found, and must be one of the following: **binary**, **ieee**, **text**, or **ascii**. The first two parameters are synonymous, as are the second two.

For **type** and the other parameters, see the preceding description of **grid**.

The syntax for the **positions** keyword is (in addition to the syntax given in 5.3, "Header File Syntax: Keyword Statements" on page 85 in *IBM Visualization Data Explorer QuickStart Guide*):

```
                           bytes n
positions =[format],[type]{ lines n   , [skip, width]  | ?, ...
                           marker string
```

(This syntax may be used only if you are specifying regular positions: origin and delta pairs for each dimension. As described in *IBM Visualization Data Explorer QuickStart Guide*, the origins and deltas are specified as origin1, delta1, origin2, delta2, etc.)

format is the format in which the grid values will be found, and must be one of the following: **binary**, **ieee**, **text**, or **ascii**. The first two parameters are synonymous, as are the second two.

For **type** and the other parameters, see the description of **grid** above. A question mark (?) signifies that the default should be used (origin=0 or delta=1) for a particular origin or delta value. This would be used if only the origins or only the deltas are to be found in the file.

B.2 Data Explorer Native Files

The Data Explorer native file format encapsulates the Data Explorer data model on disk (or on standard output as the result of an external conversion program). This file format is comprehensive and flexible in that it can represent any of the Objects created in Data Explorer. Thus, any Object can be exported at any point. A data file in this format can be imported into a Data Explorer session by specifying "dx" as the value of the **format** parameter for the Import module. For more information, see "Import" on page 165 in *IBM Visualization Data Explorer User's Reference*.

Overview of the Native File Format

A Data Explorer file consists of a header section followed by an optional data section. The header section consists of a textual description of a collection of Objects. The data section contains the Array Object data, either as text or in binary, and is referred to by the header section. A header section can refer to Objects and data either in the current file or in other files.

Figure 83 on page 245 shows data imbedded in its header. Another file cannot refer to data in this file because there is no specified data section. However, header sections in this file can refer to data sections in other files. This method is sometimes more convenient when creating data files with simple programs.

Figure 84 on page 245 shows a header section referring to a data section in another file. The header refers to the data using the data file name and an offset location (in bytes from the beginning of the data section) in the file.

Figure 85 on page 246 shows a header section and data section in the same file. The header refers to the data section using a byte offset, relative to the start of the data section.

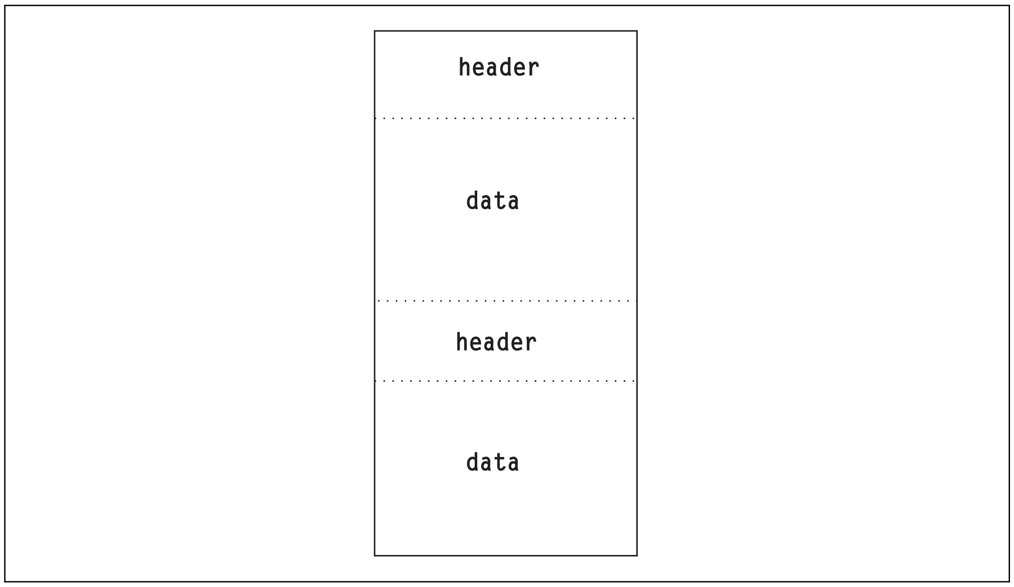


Figure 83. Data Imbedded in a Header Section

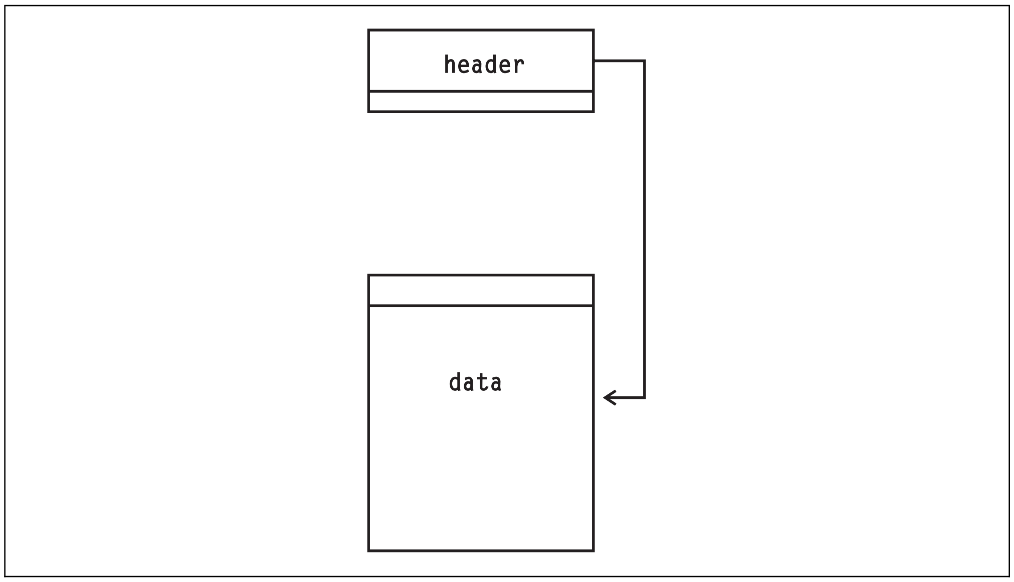


Figure 84. Header Referring to Data in Another File

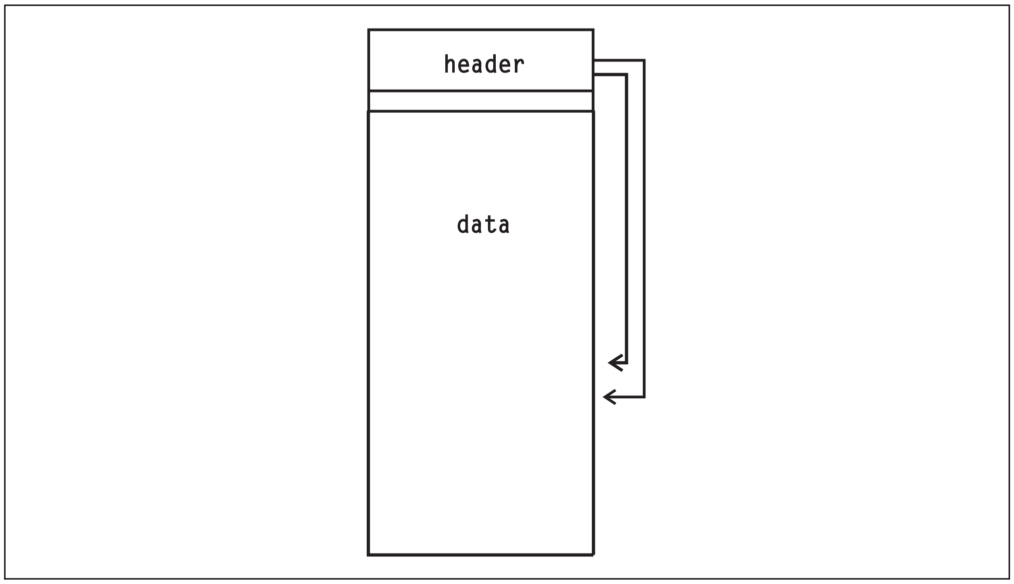


Figure 85. Header and Data in the Same File

These configurations can be used in conjunction with each other. For example, a file can contain both a header and data and can refer to data both in the same file and in another file. A file can also have only a header and refer to data in either a data-only file or in a file that contains both a header and data. This flexibility allows you to construct a header that points to data in existing files, and lets you view and edit the header information (if necessary), using standard tools.

The following examples illustrate some of the ways you can import data using the Data Explorer native file format. You may wish to refer to the full specification of the syntax (see “Syntax of the Native File Format” on page 268).

Examples

The basic way to create a data file is to first define the Arrays, or components, contained in a Field and to then describe how to collect the components together. To define a higher level structure, such as a series, first define the components, then the Fields, and then how to collect the Fields to make a series. The examples in this section illustrate the process.

In the first six examples, the data Objects can be viewed by the script shown here. Other scripts are shown with the later examples.

```
data = Import("filename.dx", format = "dx");

connections = ShowConnections(data);
connections = AutoColor(connections);

tubes = Tube(connections, 0.08);

camera = AutoCamera(tubes, "off diagonal");
image = Render(tubes, camera);
Display(image);
```

Note: For Figure 86 on page 248 and Figure 87 on page 249, the argument “off front” is used instead of “off diagonal.”

Simply substitute the file name of the data file for *filename* in the Import statement. For information about how to use the Data Explorer script language, see Chapter 10, “Data Explorer Scripting Language” on page 187.

Example 1. A Regular Grid

The following example illustrates the basic Objects of the data model, and shows how to imbed data as text in the header section. The Objects and data describe a regular grid. This file is found in **/usr/lpp/dx/samples/data/regular.dx**. Figure 86 on page 248 shows the resulting structure. The axes diagram in the lower right corner of the figure indicates the orientation of the axes. This orientation applies to all subsequent examples as well.

Note that the positions are considered to increment in the order “last index varies fastest” when matching data to positions. For example, for this simple 4 x 2 x 3 grid, the order of the positions is *[$x_0y_0z_0$]*, *[$x_0y_0z_1$]*, *[$x_0y_0z_2$]*, *[$x_0y_1z_0$]*, and so on. This is because the deltas are specified in the order *.x, y, z*, so *z* is the last index. If the data was stored in the order *[$x_0y_0z_0$]*, *[$x_1y_0z_0$] ...*, then the order of the **delta** clauses would be reversed, and the counts would be specified as 3 2 4.

When using the **gridconnections** keyword, it is not necessary to specify the “element type” or “ref” attribute, as these will automatically be set for you.

```
# This example describes a regular grid

# object 1 is the regular positions.
  The grid is 4 in x by 2 in y by 3 in z.  The origin is
# at [0 0 0], and the deltas are 1 in the first and third
# dimensions, and 2 in the second dimension
object 1 class gridpositions counts 4 2 3
origin        0            0            0
delta         1            0            0
delta         0            2            0
delta         0            0            1

# object 2 is the regular connections
object 2 class gridconnections counts 4 2 3
attribute "element type" string "cubes"
attribute "ref" string "positions"

# object 3 is the data, which is in a one-to-one correspondence with
# the positions ("dep" on positions).
# The data are matched to the positions in the order
# "last index varies fastest", i.e. (x0, y0, z0), (x0, y0, z1),
# (x0, y0, z2), (x0, y1, z0), etc.
object 3 class array type float rank 0 items 24 data follows
            1            3.4         5            2
            3.4          5.1         0.3          4.5
            1            2.3         4.1          2.1
            6            8           9.1          2.3
            4.5          5           3.0          4.3
            1.2          1.2         3.0          3.2
attribute "dep" string "positions"
```

```
# A field is created with three components: "positions", "connections",
# and "data"
object "regular positions regular connections" class field
component "positions" value 1
component "connections" value 2
component "data" value 3

end
```

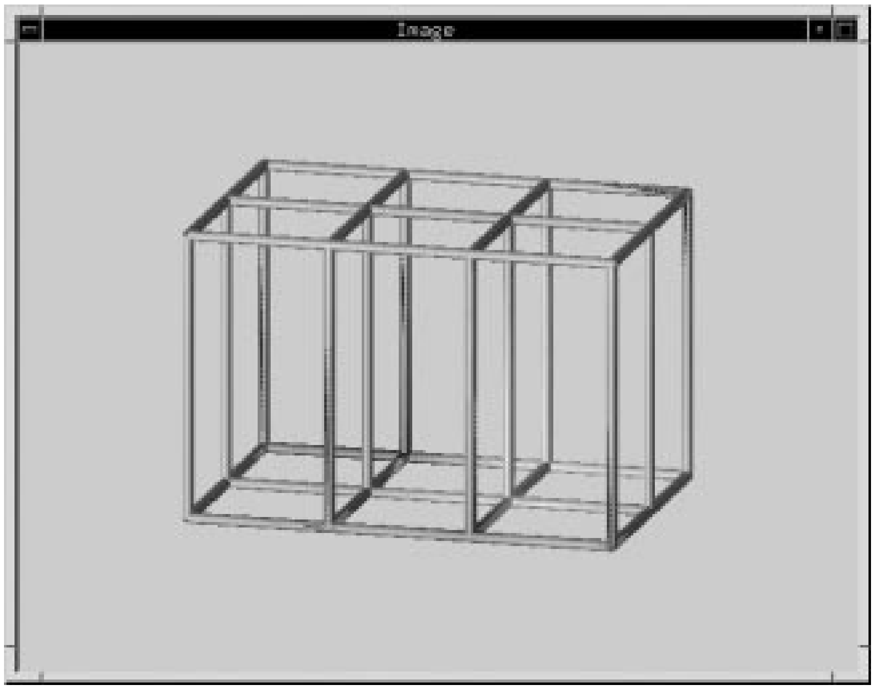


Figure 86. Regular Grid Example. The argument “off front” has been substituted for “off diagonal” in the script used to generate this figure (see “Examples” on page 246).

Example 2. A Regular Skewed Grid

This example is similar to the previous one. However, the “positions” component is changed slightly so that the Objects and data describe a regular skewed grid. This file is found in **/usr/lpp/dx/samples/data/regularskewed.dx**. Figure 87 on page 249 shows the resulting structure.

```
# This example describes a regular grid, where the axes are
# non-orthogonal

# object 1 is the regular positions, where the deltas is non-orthogonal
object 1 class gridpositions counts 4 2 3
origin          0           0          0
delta           1           0.2        0
delta           0           2          0
delta           0           0          1

# object 2 is the regular connections
object 2 class gridconnections counts 4 2 3
```

```
# object 3 is the data, which is in a one-to-one correspondence with the
# positions ("dep" on positions)
object 3 class array type float rank 0 items 24 data follows
            1         3.4           5             2
            3.4       5.1           0.3           4.5
            1         2.3           4.1           2.1
            6         8             9.1           2.3
            4.5       5             3.0           4.3
            1.2       1.2           3.0           3.2
attribute "dep" string "positions"

# the field contains three components: "positions", "connections", and
# "data"
object "regular positions regular connections" class field
component "positions" value 1
component "connections" value 2
component "data" value 3

end
```

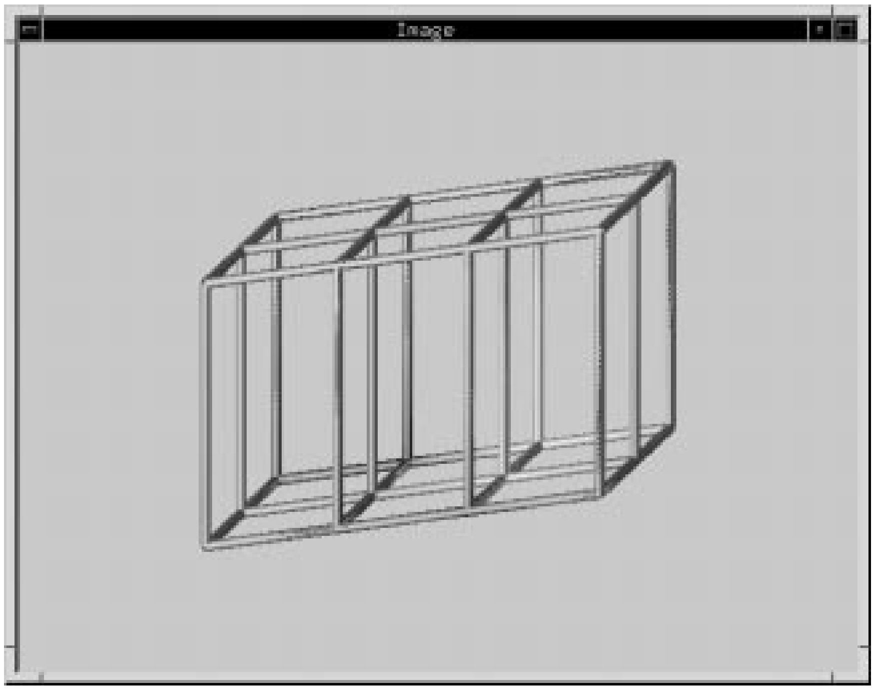


Figure 87. Regular Skewed Grid Example. The argument "off front" has been substituted for "off diagonal" in the script used to generate this figure (see "Examples" on page 246).

Example 3. A Warped Regular Grid

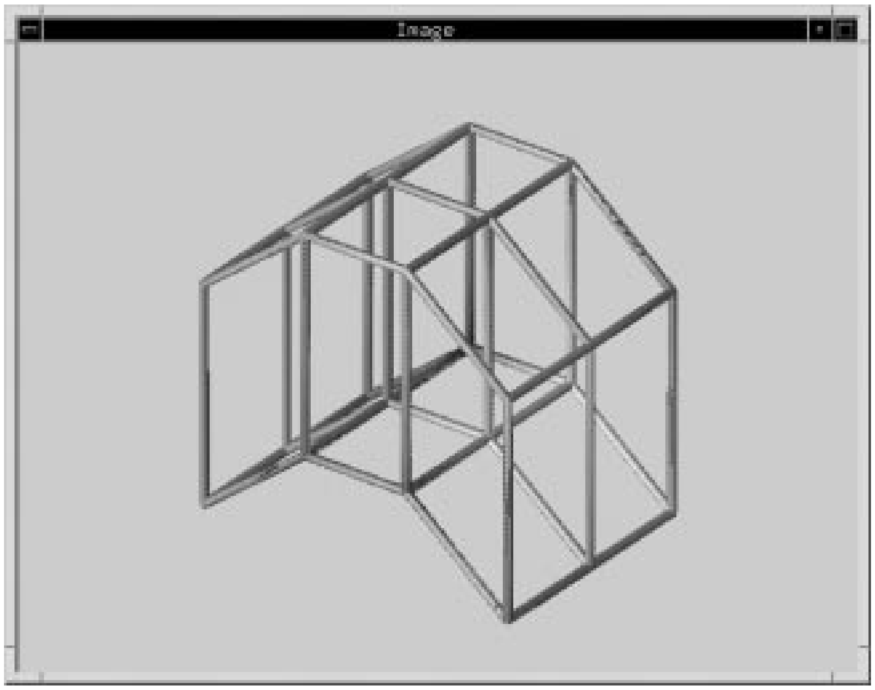


Figure 88. Warped Regular Grid Example

The example file (in **/usr/lpp/dx/samples/data/deformedregular.dx**) defines a warped regular grid and shows how to imbed data as text in a header section. The values of the “positions” component are irregular and must be enumerated. Figure 88 shows the resulting structure.

```
# The irregular, 3 dimensional positions
object 1 class array type float rank 1 shape 3 items 24 data follows
            0                0       0
            0                0       1
            0                0       2
            0                2       0
            0                2       1
            0                2       2
            1         0.841471       0
            1         0.841471       1
            1         0.841471       2
            1         2.841471       0
            1         2.841471       1
            1         2.841471       2
      2    0.9092974      0
      2    0.9092974      1
      2    0.9092974      2
      2     2.909297      0
      2     2.909297      1
      2     2.909297      2
      3      0.14112      0
      3      0.14112      1
      3      0.14112      2
      3      2.14112      0
      3      2.14112      1
      3      2.14112      2
```

```
# The regular connections
object 2 class gridconnections counts 4 2 3

# The data, in a one-to-one correspondence with the positions
object 3 class array type float rank 0 items 24 data follows
           1           3.4         5
           2           3.4         5.1
           0.3         4.5         1
           2.3         4.1         2.1
           6           8           9.1
           2.3         4.5         5
           3           4.3         1.2
           1.2         3           3.2
attribute "dep" string "positions"

# The field, with three components: "positions", "connections", and
# "data". The field is given the name "irreg positions regular connections".
object "irreg positions regular connections" class field
component "positions" value 1
component "connections" value 2
component "data" value 3

end
```

The positions are joined in the order “last index varies fastest,” with the connections specified as 4 x 2 × 3: the first 3 positions are joined in a line, as are those in each set of 3 following. Then the first 6 positions are joined as a set of 2 quadrilaterals, as are the next 6 and so on (see Figure 88 on page 250).

Example 4. An Irregular Grid

This example file (in **/usr/lpp/dx/samples/data/irregular.dx**) defines an irregular grid and shows how to imbed data as text in a header section. The values of the “positions” and “connections” components are irregular and must be enumerated. See Figure 89 on page 252.

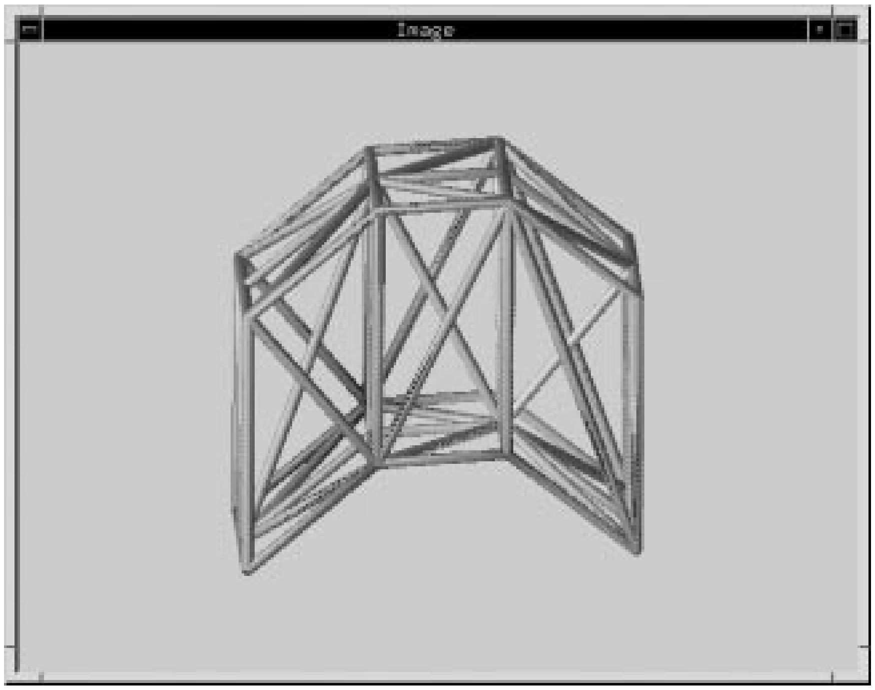


Figure 89. Irregular Grid Example

```
# The irregular positions, which are 24 3-dimensional points.
object 1 class array type float rank 1 shape 3 items 24 data follows
            0             0     0
            0             0     1
            0             0     2
            0             2     0
            0             2     1
            0             2     2
            1      0.841471     0
            1      0.841471     1
            1      0.841471     2
            1      2.841471     0
            1      2.841471     1
            1      2.841471     2
            2     0.9092974     0
            2     0.9092974     1
            2     0.9092974     2
            2      2.909297     0
            2      2.909297     1
            2      2.909297     2
            3       0.14112     0
            3       0.14112     1
            3       0.14112     2
            3       2.14112     0
            3       2.14112     1
            3       2.14112     2
```

```
# The irregular connections, which are 30 tetrahedra
object 2 class array type int rank 1 shape 4 items 30 data follows
        10        3        4        1
        3         10       9        6
        10        1        7        6
        6         1        3        10
        6         1        0        3
        10        1        4        5
        5         1        8        10
        8         5        2        1
        10        8        7        1
        5         8        11       10
        15        6        9        10
        10        6        13       15
        13        10       7        6
        15        13       12       6
        10        13       16       15
        17        10       11       8
        10        17       16       13
        17        8        14       13
        13        8        10       17
        13        8        7        10
        22        15       16       13
        15        22       21       18
        22        13       19       18
        18        13       15       22
        18        13       12       15
        22        13       16       17
        17        13       20       22

      20   17   14   13
      22   20   19   13
      17   20   23   22
attribute "element type" string "tetrahedra"
attribute "ref" string "positions"

# The data, which is in a one-to-one correspondence with the positions
object 3 class array type float rank 0 items 24 data follows
        1         3.4       5         2         3.4
        5.1       0.3       4.5       1         2.3
        4.1       2.1       6         8         9.1
        2.3       4.5       5         3         4.3
        1.2       1.2       3         3.2
attribute "dep" string "positions"

# the field, with three components: "positions", "connections", and
# "data"
object "irregular positions irregular connections" class field
component "positions" value 1
component "connections" value 2
component "data" value 3
end
```

Example 5. Header and Data in Separate Files

The following example uses a header file that contains no data. Instead, it refers to another file, **irregirreg2.bin**, that contains the data in binary format. This example contains the same information as "Example 4. An Irregular Grid" on page 251, but the data is stored in a file separate from the header. If you use this sample header file in a script, the results are the same as in Figure 89 on page 252. This file can be found in **/usr/lpp/dx/samples/data/irregirreg2.dx**.

```
object 1 class array type float rank 1 shape 3 items 24 msb binary
data file irregirreg2.bin,0
attribute "dep" string "positions"

object 2 class array type int rank 1 shape 4 items 30 msb binary
data file irregirreg2.bin,288
attribute "element type" string "tetrahedra"
attribute "ref" string "positions"

object 3 class array type float rank 0 items 24 msb binary
data file irregirreg2.bin,768
attribute "dep" string "positions"

object "irreg positions irreg connections binary file" class field
component "positions" value 1
component "connections" value 2
component "data" value 3

end
```

Often, you can use this method to point to existing data files. To do this, your header file must:

- Describe the coordinate system of the data.
- Indicate how many data values there are in the data file.
- Indicate the type of data values (float, byte, scalar, vector, and so on).

For example, suppose you have an existing data file written in the IEEE floating point format. It has the following characteristics:

- It is on a regular grid, 100 x 100 x 15, and the delta in the *z* direction is 2, while the deltas in the *x* and *y* directions are 1.
- The origin of the grid is at [50 100 10].
- The first three bytes of the file are the number of elements in the *x*, *y*, and *z* directions.
- The data values are listed in an order such that *z* varies fastest.

Given all these conditions, the following Data Explorer header file imports the data (substituting the data file name for *data_file_name*):

```
object 1 class gridpositions counts 100 100 15
origin      50     100    10
delta       1      0      0
delta       0      1      0
delta       0      0      2

object 2 class gridconnections counts 100 100 15
attribute "element type" string "cubes"
attribute "ref" string "positions"
```

```
# It skips the first three bytes before reading the data values
object 3 class array type float rank 0 items 150000
ieee data file data_file_name,3

object "field" class field
component "positions" value 1
component "connections" value 2
component "data" value 3

end
```

Example 6. Product Arrays

The following examples show how to use Product Arrays to define positions that are composed of products of arrays. Such positions may be regular in one or more dimensions and irregular in one or more dimensions. The resulting product array is found as a product of all possible combinations of the terms comprising the Array.

The first data file defines data that has irregular positions in the xy plane but regular spacing in the z dimension. This file can be found in **/usr/lpp/dx/samples/data/product1.dx**. Figure 90 on page 256 shows the resulting image.

```
# define a set of irregular points in the xy plane
object 1 class array type float rank 1 shape 3 items 8 data follows
        0.0    0.0    0.0
        0.0    1.1    0.0
        1.0    0.2    0.0
        1.1    1.3    0.0
        2.2    0.2    0.0
        2.5    1.1    0.0
        3.5    0.1    0.0
        3.4    1.0    0.0

# define a set of regular points in the z direction
object 2 class regulararray count 3
origin     0.0      0.0      0.0
delta      0.0      0.0      1.0

# create a product array of the irregular points in the xy plane and
# the regular points in the z direction
object 3 class product array
    term 1
    term 2

# create regular cube connections
object 4 class gridconnections counts 4 2 3

# the data component
object 5 class array type float rank 0 items 24 data follows
      1.0      2.1      2.0      1.0      4.5      6.7      8.1      2.0
     -0.9     -0.8      1.0      1.2      1.3      0.1      0.3      3.0
      1.2      3.2      4.1      0.9      2.0      1.0     -0.9      2.0

object "field" class field
     component "positions" 3
     component "connections" 4
     component "data" 5

end
```

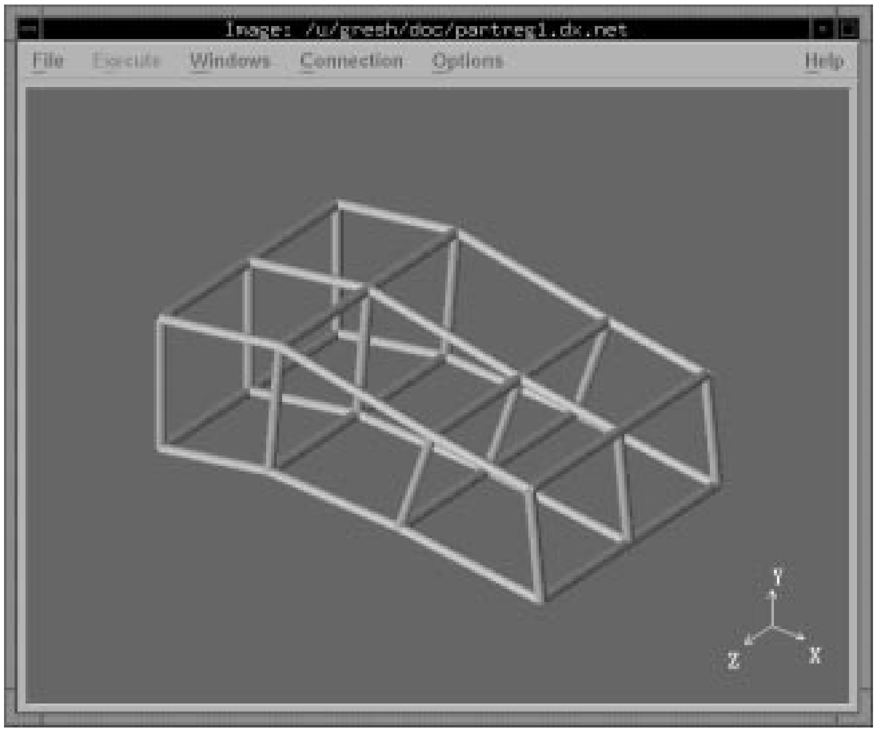


Figure 90. Product Array Example with Irregular Points in the XY Plane

The next data file defines a regular grid that is regular in the xy plane but has irregular positions in the z direction. This file can be found in **/usr/lpp/dx/samples/data/product2.dx**. Figure 91 on page 257 shows the resulting image.

```
# define a set of regular points in the xy plane
object 1 class gridpositions 4 2 1

# define a set of irregular points in the z direction
object 2 class array type float rank 1 shape 3 items 3 data follows
      0.0       0.0       0.0
      0.0       0.0       1.0
      0.0       0.0       3.0

# create a product array of the regular points in the xy plane and
# the irregular points in the z direction
object 3 class product array
      term 1
      term 2

# create regular cube connections
object 4 class gridconnections counts 4 2 3

# the data component
object 5 class array type float rank 0 items 24 data follows
        1.0      2.1    2.0    1.0     4.5     6.7      8.1      2.0
       -0.9     -0.8    1.0    1.2     1.3     0.1      0.3      3.0
        1.2      3.2    4.1    0.9     2.0     1.0     -0.9      2.0
```

```
object "field" class field
       component "positions" 3
       component "connections" 4
       component "data" 5

end
```

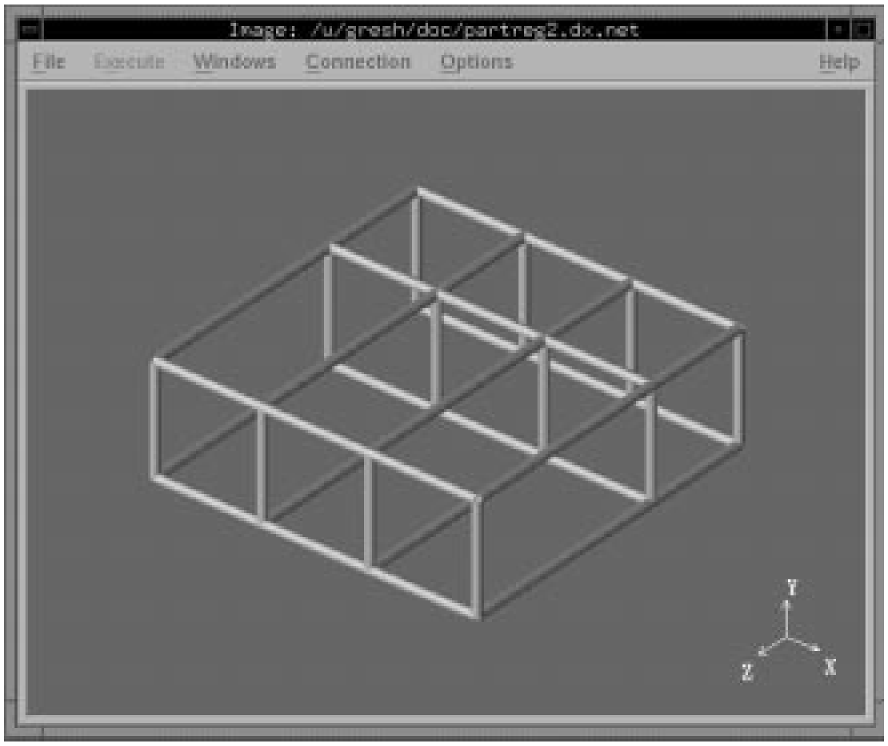


Figure 91. Product Array Example with Irregular Points in the Z Direction

Example 7. Series

The following file defines the data for a series. It defines three data Array Objects, and then three Field Objects that are associated with the data. The grid definitions are in a separate file (**pos_conn.data**). This first file can be found in **/usr/lpp/samples/data/regseries.dx**.

```
# This example describes a data series with three member fields
# Object 1 is the data associated with the first frame in the
# series. The data is "dep" positions, or in a one-to-one
# correspondence with positions.
object 1 class array type float rank 1 shape 3 items 18 data follows

     1.0    0.1    0.0
     1.4    0.2    0.0
     1.0    0.0    0.0
     2.2    0.1    0.2
     1.0    0.0    0.0
     2.0    0.0    0.1
     0.9    0.1    0.0
     1.1   -0.4    0.0
     1.0    0.1    0.0
     1.2    0.1    0.1
     0.3    0.0    0.0
```

```
      1.0       0.1       0.1
      1.1      -0.4       0.2
      1.1       0.2       0.0
      1.1       0.1       0.0
      1.2       0.1       0.1
      1.1       0.0       0.0
      0.9       0.0       0.1
attribute "dep" string "positions"

# Object 2 is the data associated with the second frame in the series.
object 2 class array type float rank 1 shape 3 items 18 data follows
      0.0       1.1       0.0
      0.1       2.2       0.0
      0.0       1.0       0.0
      0.2       1.1      -0.2
      0.0       0.8       0.0
      0.0       1.9       0.4
      0.1       1.1       0.0
      0.1       1.2       0.0
      0.0       1.1       0.0
      0.2       2.1       0.1
      0.1       1.0       0.0
      0.0       0.8       0.1
      0.1       0.9      -0.2
      0.1       1.2       0.0
      0.1       1.1       0.0
      0.2       1.1      -0.4
      0.1       0.9       0.0
      0.2       0.9       0.1
attribute "dep" string "positions"

# Object 3 is the data associated with the third frame in the series.
object 3 class array type float rank 1 shape 3 items 18 data follows
      0.0       0.1       1.0
      0.1       0.2       1.0
      0.0       0.0       2.0
     -0.2       0.1       2.2
      0.0       0.1       0.9
      0.0       0.2       0.8
     -0.4       0.1       0.9
      0.1       0.2       1.9
      0.0       0.1       0.7
      0.2       0.1       1.1
     -0.1       0.0       2.0
      0.0       0.3       1.1
      0.1       0.1       1.2
     -0.5       0.2       1.0
      0.1       0.1       2.0
      0.2       0.1       1.4
      0.1      -0.3       0.9
      0.2       0.3       1.1
attribute "dep" string "positions"
```

```
# Object 4 is the first field in the series. The positions and
# connections are defined by objects 1 and 2 in a separate file,
# "pos_conn.data", and the data is given by object 1 in this file.
object 4 class field
component "positions" value file "pos_conn.data",1
component "connections" value file "pos_conn.data",2
component "data" value 1

# Object 5 is the second field in the series. The positions and
# connections are defined by objects 1 and 2 in a separate file,
# "pos_conn.data" (and are in fact the same positions and connections
# as those of the first field), and the data is given by object 2 in this file.
object 5 class field
component "positions" value file "pos_conn.data",1
component "connections" value file "pos_conn.data",2
component "data" value 2

# Object 6 is the third field in the series. The positions and
# connections are defined by objects 1 and 2 in a separate file,
# "pos_conn.data" (and are in fact the same positions and connections
# as those of the first field), and the data is given by object 3 in
# this file.
object 6 class field
component "positions" value file "pos_conn.data",1
component "connections" value file "pos_conn.data",2
component "data" value 3

# Here we create the series object with three members.
# The members are objects 4, 5, and 6, which we defined above.
# Each has a position tag associated with it (for example a time tag).
object "series" class series
member 0 value 4 position 1.3
member 1 value 5 position 2.5
member 2 value 6 position 4.5

end
```

The following file defines the grid for this time series. This file can be found in **/usr/lpp/samples/data/pos_conn.data**.

```
object 1 class gridpositions counts 3 2 3
origin          0            0           0
delta           1            0           0
delta           0            2           0
delta           0            0           1

object 2 class gridconnections counts 3 2 3
attribute "element type" string "cubes"
attribute "ref" string "positions"

end
```

Example 8. Two-dimensional Grid, Cell-centered Data

This example describes a regular 2-dimensional grid. In this example, unlike other examples presented here, the data are dependent on (in a one-to-one correspondence with) the "connections" rather than the "positions" component. Data Explorer interprets this as implying that the data value within each connection element is the constant given by the corresponding data value. For example, if you used AutoColor and rendered this Field, you would see blocks of constant color.

You can use the following script to render this Object:

```
  data = Import("/usr/lpp/samples/data/datadepconnections.dx", format="dx");
  colored = AutoColor(data);
  camera = AutoCamera(colored);
  Display(colored, camera);
```

The data file is located in **/usr/lpp/samples/data/datadepconnections.dx**.

```
# object 1 is the regular positions. The grid is 4x4. The origin is
# at [0 0], and the deltas are 1 in the first dimension, and
# 2 in the second dimension
object 1 class gridpositions counts 4 4
origin     0    0
delta      1    0
delta      0    2

# object 2 is the regular connections, quads, connecting the positions
object 2 class gridconnections counts 4 4

# object 3 is the data, which are in a one-to-one correspondence with
# the connections ("dep" on connections)
object 3 class array type float rank 0 items 9 data follows
           1          3.4           5           2
           3.2        5.5           0.3         4.5
           4.0
attribute "dep" string "connections"

# A field is created with three components: "positions", "connections",
# and "data"
object "regular positions regular connections" class field
component "positions" value 1
component "connections" value 2
component "data" value 3

end
```

Example 9. Faces, Loops, and Edges

Faces loops, and edges are used to define polygons. For example, you may wish to define regions of a map with polygons. A positions component identifies the vertices of the polygons, an edges component identifies how to connect positions, a loops component identifies the beginning of each loop by referring to the first edge of the loop, and a faces component identifies which loops make up a face. (A face may have more than one loop if the face has one or more holes in it.)

For more information about faces, loops, and edges, see "Faces, Loops, and Edges Components" on page 24. Note that some modules do not accept this kind of data. However, the Refine module can be used to convert faces, loops, and edges data to triangles. See “Refine” on page 258 in *IBM Visualization Data Explorer User’s Reference*.

The following example describes a simple 2-dimensional data set consisting of five polygons. None of the polygons has holes. To view the data, you can use the following script:

```
g = Import("FacesLoopsEdges.dx");
c = AutoCamera(g);
colored = AutoColor(g);
Display(colored, c);
```

The data file is given below, and the resulting connections are illustrated in Figure 92 on page 263. The data file can be found in **/usr/lpp/dx/samples/data/FacesLoopsEdges.dx**.

```
#
# Example of faces, loops and edges components.
# This example has no holes.
#

#
# Positions array.  These are a list of all the vertices of the object;
#   no particular order is required here.
#
object "position list" class array type float rank 1 shape 2 items 11
data follows
    0.133985      0.812452      # point number 0
    0.375019      0.896258      # point number 1
    0.532733      0.76484       # point number 2
    0.523806      0.404777      # point number 3
    0.300626      0.327407      # point number 4
    0.145888      0.508927      # point number 5
    0.68152       0.851137      # point number 6
    0.815428      0.758889      # point number 7
    0.94636       0.592248      # point number 8
    0.729132      0.416679      # point number 9
    0.535709      0.190524      # point number 10
#
#  Edges array.  This is a list of connected points, by point number.
#  All the edges associated with a particular face need to be listed
#  together. If points 10, 3 and 7 make a triangle, the list is
#  "10 3 7" and the 10 is not repeated. Note that below, for
#  readability, the connected points for each loop are
#  shown together. However line breaks are not significant
#  to the importer, and all of the following numbers could have
#  been on the same line, or one to a line, with the same result.
object "edge list" class array type int rank 0 items 21 data follows
    1     2     6                               # edge point index 0
    0     5     4     3     2     1             # edge point index 3
    2     3     9     8     7     6             # edge point index 9
    3    10     9                               # edge point index 15
    3     4    10                               # edge point index 18
attribute "ref" string "positions"
```

```
#  Loops array.  This is a list of connected edges, by edge number.
#  Each number is the edge index of where the next loop starts.
object "loop list" class array type int rank 0 items 5 data follows
      0     # loop index 0
      3     #  1
      9     #  2
     15     #  3
     18     #  4
attribute "ref" string "edges"

#
#  Faces array.  This is list of which loops make faces.  If there are
#   no holes in the faces, this is list of all loops.  If two or more
#   loops actually describe the outside edges and inside hole edges of
#   a face, then this list contains the starting loop numbers of the
#   list of loops making up a face.
#
object "face list" class array type int rank 0 items 5 data follows
 0
 1
 2
 3
 4
attribute "ref" string "loops"

#  data array. Dependent on faces.
#
object "data" class array type float rank 0 items 5 data follows
    0     2.5        1.2   0.4   1.8
attribute "dep" string "faces"

#
#  Field definition to put the arrays together.
#
object "map" class field

       component "positions"     "position list"
       component "edges"         "edge list"
       component "loops"         "loop list"
       component "faces"         "face list"
       component "data"          "data"

end
```

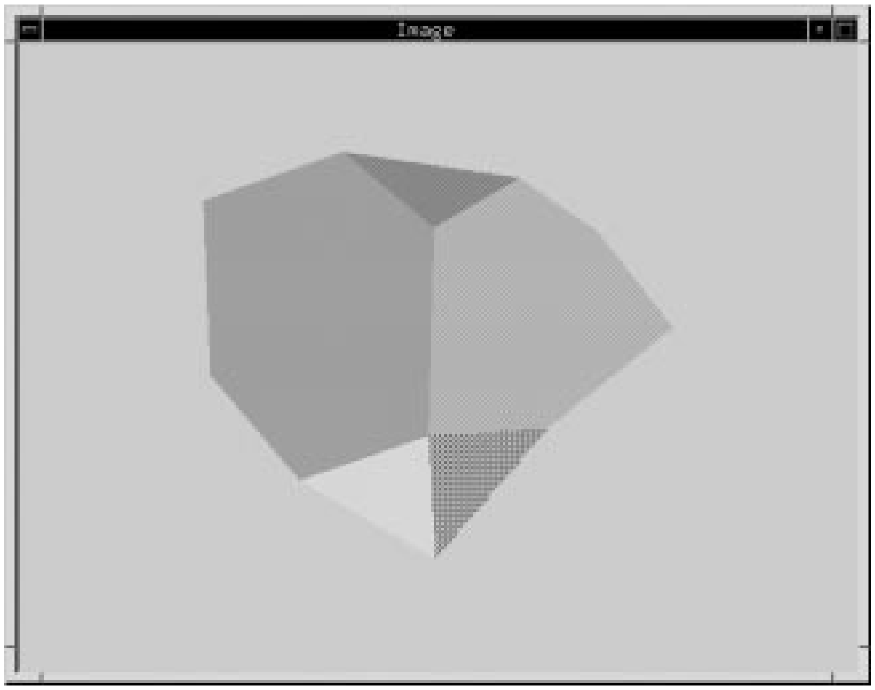


Figure 92. Example of Faces, Loops, and Edges

Example 10. Faces, Loops, and Edges with a Hole

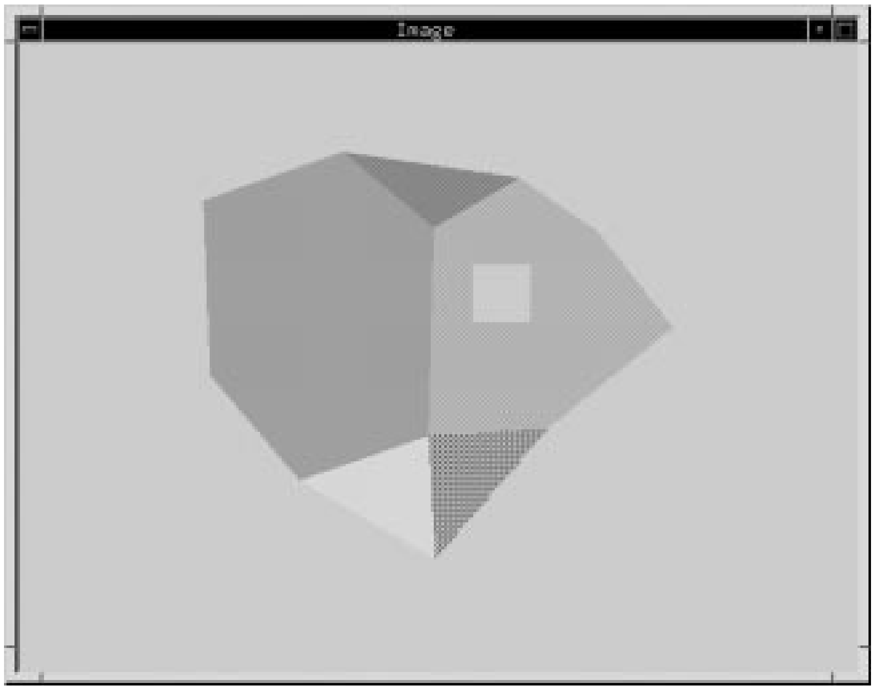


Figure 93. Example of Faces, Loops, and Edges with a Hole

The following data file is identical to the previous data file except that the third polygon has a square hole in it. The resulting connections are illustrated in Figure 93.

```
#
# Example of faces, loops and edges components. The third face has a
# square hole in it.
#
#
# Positions array.  These are a list of all the vertices of the object;
#   no particular order is required here.
#
object "position list" class array type float rank 1 shape 2 items 15
data follows
    0.133985      0.812452      # point number 0
    0.375019      0.896258      # point number 1
    0.532733      0.76484       # point number 2
    0.523806      0.404777      # point number 3
    0.300626      0.327407      # point number 4
    0.145888      0.508927      # point number 5
    0.68152       0.851137      # point number 6
    0.815428      0.758889      # point number 7
    0.94636       0.592248      # point number 8
    0.729132      0.416679      # point number 9
    0.535709      0.190524      # point number 10
    0.600000      0.700000      # point number 11
    0.700000      0.700000      # point number 12
    0.700000      0.600000      # point number 13
    0.600000      0.600000      # point number 14
#
#  Edges array.  This is a list of connected points, by point number.  All
#  the edges associated with a particular face need to be listed together.
#  If points 10, 3 and 7 make a triangle, the list is "10 3 7" and the 10
#  is not repeated.  Following a polygon, the front of the polygon is
#  determined by the right-hand rule.
object "edge list" class array type int rank 0 items 25 data follows
    1     2     6                       # edge point index 0
    0     5     4     3     2     1       # edge point index 3
    2     3     9     8     7     6       # edge point index 9
   11    12    13    14                 # edge point index 15 (hole in third face)
    3    10     9                       # edge point index 19
    3     4    10                       # edge point index 22
attribute "ref" string "positions"
#
#  Loops array.  This is a list of connected edges, by edge number.  Each
#  number is the edge index of where the next loop starts.
object "loop list" class array type int rank 0 items 6 data follows
      0     # loop index 0
      3     #  1
      9     #  2
     15     #  3
     19     #  4
     22     #  5
attribute "ref" string "edges"
```

```
#
#  Faces array.  This is list of which loops make faces.  If there are
#   no holes in the faces, this is list of all loops.  If two or more
#   loops actually describe the outside edges and inside hole edges of
#   a face, then this list contains the starting loop numbers of the
#   list of loops making up a face.
#
object "face list" class array type int rank 0 items 5 data follows
 0
 1
 2
 4
 5
attribute "ref" string "loops"

#
#  data array. Dependent on faces.
#
object "data" class array type float rank 0 items 5 data follows
   0     2.5        1.2   0.4   1.8
attribute "dep" string "faces"

#
#
#  Field definition to put the arrays together.
#
object "map with hole" class field
       component "positions"     "position list"
       component "edges"         "edge list"
       component "loops"         "loop list"
       component "faces"         "face list"
       component "data"          "data"
end
```

Example 11. Three-dimensional Faces, Loops, and Edges

The following data file describes a faces, loops, and edges Object that exists in 3-dimensional space.

```
// To view the solid:
g = Import("solid.dx");
a = AutoCamera(g, [0.2 0.1 1.0]);
Display(g, a);

// To look at just the connections:
s = ShowConnections(g);
Display(s, a);
```

Use the first part of the script to view the solid, and the second part to view only the connections. The data file is given below, and the resulting connections are shown in Figure 94 on page 266. This data file can be found in **/usr/lpp/samples/data/solid.dx**.

```
# Example of faces, loops and edges components.
#
# Positions array.  These are a list of all the vertices of the object;
#   no particular order is required here.
object "position list" class array type float rank 1 shape 3 items 12
data follows
```

```
    5.0    0.0    1.0        # point number 0
    5.0    0.0    5.0        # 1
    3.0    5.0    5.0        # 2
    4.5    0.0    1.5        # 3
    4.5    0.0    4.5        # 4
    3.0    4.0    5.0        # 5
    4.0    0.5    5.0        # 6
    1.5    0.0    1.5        # 7
    1.5    0.0    4.5        # 8
    2.0    0.5    5.0        # 9
    1.0    0.0    5.0        # 10
    1.0    0.0    1.0        # 11

#  Edges array.  This is a list of connected points, by point number.  All
#  the edges associated with a particular face need to be listed together.
#  If points 10, 3 and 7 make a triangle, the list is "10 3 7" and the 10
#  is not repeated.  If there is a hole in the triangle, the edges that
#  describe the hole must be listed right before or right after.
object "edge list" class array type int rank 0 items 23 data follows
    1     0     2             # edge point index 0
   10     1     2             # 3
    9     6     5             # 6
   10     1     0    11        # 9
    8     4     3     7        # 13
   11    10     2             # 17
    0    11     2             # 20
attribute "ref" string "positions"

#  Loops array.  This is a list of connected edges, by edge number.  Each
#   number is the edge index of where the next loop starts.
object "loop list" class array type int rank 0 items 7 data follows
```

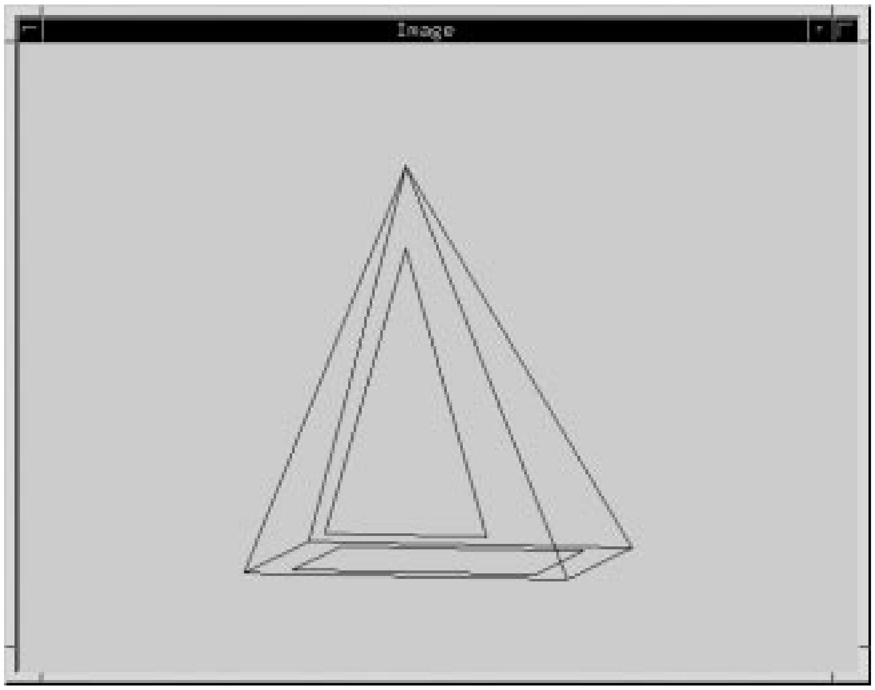


Figure 94. Example of a Surface Using Faces, Edges, and Loops

```
     0     # loop index 0
     3     # 1
     6     # 2
     9     # 3
     13    # 4
     17    # 5
     20    # 6
attribute "ref" string "edges"

#  Faces array.  This is list of which loops make faces.  If there are
#   no holes in the faces, this is list of all loops.  If two or more
#   loops actually describe the outside edges and inside hole edges of
#   a face, then this list contains the starting loop numbers of the
#   list of loops making up a face.
object "face list" class array type int rank 0 items 5 data follows
 0
 1
 3
 5
 6
attribute "ref" string "loops"

#  Colors array.  To get flat shaded surfaces, there should be one color
#    per face, and one normal per face.  These are Red,Green,Blue values
#    between 0 (no color) and 1 (fully saturated).
object "color list" class array type float rank 1 shape 3 items 5 data follows
    0.6     0.3     0.6
    0.8     0.8     0.1
    0.9     0.4     0.9
    0.4     0.8     0.7
    0.8     0.8     0.8
attribute "dep" string "faces"

#  Normals array.
object "normal list" class array type float rank 1 shape 3 items 5
data follows
    0.93      0.37     0.0
    0.0       0.0      1.0
    0.0      -1.0      0.0
   -0.93      0.37     0.0
    0.0      -0.63    -0.78
attribute "dep" string "faces"

#  Field definition to put the arrays together.
object "solid" class field
       component "positions"     "position list"
       component "edges"         "edge list"
       component "loops"         "loop list"
       component "faces"         "face list"
       component "colors"        "color list"
       component "normals"       "normal list"

end
```

Example 12. Image Files

This Data Explorer header file reads an image (**`cylinder.rgb`**). The image is 350 x 300 and consists of RGB colors (3-vectors). You can read this image in, and display it, using the visual program **`/usr/lpp/dx/samples/programs/ReadImage.net`**. This file can be found in **`/usr/lpp/dx/samples/data/image.dx`**.

Note: It is easier to import a file in the RGB format by using the ReadImage module. This example illustrates the different aspects of header files.

```
   # First describe the positions. The image is written such that
   # x varies fastest, and the first pixel in the file is the one that is
   # to be displayed at the top left.
   # Because x varies fastest, the last delta specifies a vector in
   # the x direction.  Because the first pixel is at the top left,
   # the delta in the y direction is -1.
object 1 class gridpositions 300 350
origin      0   0
delta       0  -1
delta       1   0

   # Next describe the connections
   # The image is 350 pixels in x and 300 pixels in y.  Since
   # x is the last delta specified, the connections are specified as
   # 300 x 350
object 2 class gridconnections 300 350
attribute "ref" string "positions"
attribute "element type" string "quads"

   # Next indicate that the data can be found in the file "cylinder.rgb",
   # starting at byte 0.  There are three bytes (red, green, and blue)
   # for each pixel.
object 3 class array type byte rank 1 shape 3 ieee msb items 105000
data file cylinder.rgb,0
attribute "dep" string "positions"

   # We read the colors in as the "data" component. This allows us
   # to immediately begin operating on them (for example, to convert the
   # bytes to floating point colors)
object "image" class field
component "positions" 1
component "connections" 2
component "data" 3
```

Syntax of the Native File Format

A data file in the Data Explorer native format can contain a header section, a data section, or both. The header section defines a set of Objects, the data for which are contained in the data section or imbedded in the header. Each type of object that can be defined is described in the following subsections.

Header Section

The header section of a Data Explorer file consists of a sequence of Object definitions. Each Object definition consists of a sequence of clauses, beginning with an **`object`** clause. The clauses defining an Object can be in any order, except that the type and size information for an Array must be specified before the data if the data are imbedded in the header. A clause consists of a sequence of words separated by one or more blank spaces or new lines. Line breaks are not significant (except after the **`follows`** keyword, when data must follow on the next line). Multiple clauses can occur on one line, and a single clause can be split across lines. The following sections describe each of the types of Objects that can be defined. In these descriptions, the **monospace** font specifies literals; *italics*, non-literals; square brackets [ ], optional items; and a vertical list, alternatives.

In the header section (and text data sections), **#** is the comment character. All text from **#** to the end of the line is ignored.

Data Section

The data section is used for Array data when either an offset in the current file or in a separate file is specified in the Array Object definition. All other Objects, including Array Objects whose definitions use the **'data follows'** specification, are self-contained; their definitions include all necessary information. These Objects do not use a data section.

The Data Explorer file format is flexible enough to describe many existing data formats without having to reformat the data. It allows you to specify byte order, which index varies fastest, whether the data type is floating point or byte, and whether the file format is binary or ASCII.

For data that is not in a format accepted by the Data Explorer native file format, you can either reformat the data so it is acceptable, or import the data using a general importer specification (see 5.1, “General Array Importer” on page 63 in *IBM Visualization Data Explorer QuickStart Guide*).

The data section consists of a set of data items specified as text or binary by the **data** clauses in the various Array Object definitions. Text and binary can be mixed in the same data section, because the **data** clauses specify portions of the data section by byte offsets. Binary representations can be in most-significant-byte-first (**msb**) or least-significant-byte-first (**lsb**) format, as specified by the relevant **data** or **'data mode'** clause. Binary floating-point numbers currently must be in IEEE format.

If a Data Explorer file does not begin with a valid header (at least an **end** clause), it is assumed not to have a header section.

Objects

Every Object has a class and an identifying numeric or string name whose scope is the file containing the Object. The definition of an Object is introduced by an **object** clause that specifies the Object number or name and its class. The **class** keyword is optional (as are many of the keywords in the following list).

```
object  number [class]     group
        "name"             series
                           multigrid
                           compositefield
                           field
                           array
                           constantarray
                           gridpositions
                           regulararray
                           productarray
                           gridconnections
                           patharray
                           mesharray
                           xform
                           string
                           light
                           camera
                           clipped
                           screen
```

The numeric or string name of an Object is used to refer to that Object in the definitions of other Objects. In general such references can take one of several forms:

```
number
"name"
file file
file file,number
file file,"name"
```

An Object that has a string name can be imported with the Import module by specifying that string as the **variable** parameter of Import.

All Objects can have any number of named attributes specified by **attribute** clauses. The value of an attribute is an Object. The value can be specified as a string or list of strings by using the **string** keyword (in which case a String Object is created to hold the string), as a number by using the **number** keyword (in which case an Array Object is created to hold the number), or as an Object reference.

```
attribute "attribute-name" [value]      string  "string" ...
                                        number number
                                        number
                                        "name"
                                        file file
                                        file file,number
                                        file file,"name"
```

Group Objects

A Group Object has any number of named or numbered members specified by **member** clauses. The value of the member is specified as an Object name or number in the current file or as an Object name or number in another file. Member numbers must be sequential, starting at 0, with no gaps in the numbering.

```
object   number [class] group
         "name"
member   "member-name" [value]        number
         number                       "name"
                                      file file
                                      file file,number
                                      file file,"name"
```

Series Objects

A Series Object is a subclass of Group Object, in which each member has in addition to its ordinal index a floating-point *series position.* The series position can be, for example, the time stamp for each member in a time series. A Series Object has any number of numbered members. The value of the member is specified as an Object name or number within the current file or as an Object name or number in another file. Member numbers must be sequential, starting at 0, with no gaps in the numbering.

```
object   number [class] series
         "name"
member   number [position] number [value]     number
                                              "name"
                                              file file
                                              file file,number
                                              file file,"name"
```

Multigrid Objects

A Multigrid Object is a subclass of Group Object, in which each member is constrained to have the same data and connections type. This is used for representing a Field as a collection of primitive Fields. Fields may be spatially disjoint or they may overlap. A Multigrid Object has any number of named or numbered members specified by **member** clauses. The value of the member is specified as an Object name or number within the current file or as an Object name or number in another file. Member numbers must be sequential, starting at 0, with no gaps in the numbering.

```
object   number [class] multigrid
         "name"
member   "member-name" [value]       number
         number                      "name"
                                     file file
                                     file file,number
                                     file file,"name"
```

Composite Field Objects

A Composite Field Object is a subclass of Group Object, in which each member is constrained to have the same data and connections type. In addition, Fields must be spatially disjoint and abutting, with boundary positions replicated exactly. A Composite Field Object has any number of named or numbered members specified by **member** clauses. The value of the member is specified as an Object name or number in the current file or as an Object name or number in another file. Member numbers must be sequential, starting at 0, with no gaps in the numbering.

```
object   number [class] compositefield
         "name"
member   "member-name" [value]       number
         number                      "name"
                                     file file
                                     file file,number
                                     file file,"name"
```

Field Objects

A Field Object has any number of named components specified by **component** clauses. The value of the component is specified as an object name or number in the current file or as an Object name or number in another file.

```
object       number [class] field
             "name"
component    "component-name" [value]      number
                                           "name"
                                           file file
                                           file file,number
                                           file file,"name"
```

Array Objects

An Array Object specifies the type (default `float`), category (default `real`), rank (default 0), shape, and number of items. The types are defined as follows:

signed byte
unsigned byte
signed 2-byte integer
unsigned 2-byte integer
signed 4-byte integer
unsigned 4-byte integer
signed 8-byte integer
4 byte floating point
8 byte floating point
character string

Note: Lists are simply Arrays. Thus a list of integers is an Array of type "int"; a list of strings, an Array of type "string."

The categories are:

real—A number
complex—Two numbers representing the real and imaginary components.

The data is specified by an offset in bytes in the data section of the current file, an offset within the data section of another Data Explorer file, or by the keyword **follows**, indicating that the data begins immediately following the newline after the **follows** keyword. The offset is specified in bytes for both binary and text files.

Optional keywords before the **data** keyword specify the format and byte order of the data. The **mode** keyword before a data-location specification sets the default data encoding for all subsequent **data** clauses to be the most recently defined data encoding. The default data encoding is **text** (or **ascii** on all currently supported systems). The **ieee** keyword specifies the ANSI/IEEE standard 754 data format.

If **binary** (or **ieee** on all currently supported systems) is specified, the default byte order depends on the platform on which Data Explorer is running. On the DEC

Alpha, the default byte order is **lsb** (least significant byte first). On all other platforms, the default byte order is **msb** (most significant byte first). The **'data mode'** clause can be used outside an Array Object definition; see “Data Mode Clause” on page 278 for more information.

```
object          number [class] array
                “name”
[type           [unsigned] byte ]
                signed byte
                unsigned short
                [signed] short
                unsigned int
                [signed] int
                hyper
                float
                double
                string
[category       real     ]
                complex
[rank number]
[shape number ...]
items number
[  msb  ] [     text ] data [ mode ]    offset
   lsb          ieee                    file file,offset
                binary                  follows
                ascii
```

If **byte**, **short**, or **int** are not prefixed with either signed or unsigned, by default, bytes are unsigned, shorts are signed and ints are signed. For compatibility with earlier versions, **char** is accepted as a synonym for **byte**.

Note: For string-type data, the Array rank should be 1 and the Array shape should be the length of the longest string plus 1.

Constant Array Objects

A Constant Array defines an Array whose elements all have the same value.

```
object          number [class] constantarray
                “name”
[type           [unsigned] byte ]
                signed byte
                unsigned short
                [signed] short
                unsigned int
                [signed] int
                hyper
                float
                double
                string
[category       real     ]
                complex
[rank number]
[shape number ...]
items number
[  msb  ] [     text ] data [ mode ]    offset
   lsb          ieee                    file file,offset
                binary                  follows
```

ascii

If **byte**, **short**, or **int** are not prefixed with either signed or unsigned, by default, bytes are unsigned, shorts are signed and ints are signed. For compatibility with earlier versions, **char** is accepted as a synonym for **byte**.

Note: For string type data, the Array rank should be 1 and the Array shape should be the length of the string plus 1.

The only difference between the specification of a Constant Array and the specification of an Array is that for a Constant Array only one value is listed in the data section.

gridpositions Keyword

The **gridpositions** keyword is used to represent an *n*-dimensional grid of geometrically regular points in a compact form. It is a kind of Array Object and can be used in any context where an Array Object would be used. It is typically used as a regular positions component. The shape of the grid (number of points in each dimension) is specified by a list of *n* numbers following the optional **counts** keyword in the **object** clause. The number *n* of items in this list determines the dimensionality of the grid. The last item in this list corresponds to the fastest varying dimension.

A grid has an origin, which can be specified by an **origin** clause (which lists the *n* coordinates of the origin). If the **origin** clause is not present, the **origin** defaults to 0. The **origin** clause can be followed by *n* **delta** clauses, listing the deltas for each dimension. Each **delta** clause has *n* elements. The last **delta** clause corresponds to the fastest varying dimension. "Example 1. A Regular Grid" on page 247 shows how to use the **delta** clause to specify a grid in which *z* varies fastest. If the **delta** clauses are not specified, the deltas default to unit vectors in each dimension, with the last dimension varying fastest.

```
object  number [class]  gridpositions [counts]number...
        "name"
origin  number ...
delta   number ...
[ delta  number ... ]
.
:
```

The **gridpositions** keyword does not actually correspond to a primitive Object type in the system, but instead is a convenient way of representing an important special case of the more general product (Product Array Object) of *n* 1-dimensional Regular Arrays (Regular Array Objects). For most purposes the **gridpositions** keyword is sufficient and more convenient. The more primitive Regular and Product Arrays are described next.

Regular Array Objects

A Regular Array Object is a compact encoding of a linear sequence of equally spaced points in *n*-space. It is often combined in a Product Array with other Regular Array Objects to obtain a grid of equally-spaced points in *n*-space; in such a case, it is generally more convenient to use the **gridpositions** keyword described earlier.

A Regular Array Object is a linear sequence of points in *n*-space starting at some origin, specified by the **origin** clause, and separated by some constant delta, specified by the **delta** clause. The **delta** clause has the same number of elements as **origin**. The dimensionality of the space that the linear sequence is embedded in is determined by the number of coordinates specified in the **origin** clause. Regular Array Objects are always of rank 1.

```
object   number [class] regulararray [items] number
         "name"
[type    unsigned byte ]
         signed byte
         unsigned short
         signed short
         unsigned int
         signed int
         hyper
         float
         double
         string
origin   number ...
delta    number ...
```

Product Array Objects

A Product Array is a compact encoding of a generalized notion of a regular grid. It is frequently used to describe a rectilinear grid as a product of Regular Arrays; in such a case, it is generally more convenient to use the **gridpositions** keyword described earlier.

A Product Array is the set of all possible sums of the points of the terms forming the product. For example, the product of a set of Arrays, each of which is a Regular Array as described above, is a lattice of points with basis vectors equal to the deltas of the Regular Arrays and origin equal to the sum of the origins of the terms. A product of a Regular Array with an irregular Array is a "semi-regular" grid whose unit cells are prisms. A Product Array is specified by a list of **term** clauses naming the Arrays that form the product. The last term varies fastest in the resulting list of positions.

```
object   number [class] productarray
         "name"
term     number
         "name"
         file file
         file file,number
         file file,"name"
```

gridconnections Keyword

The **gridconnections** keyword is used to represent the *n*-dimensional cuboidal connections of a regular grid in a compact form. It is a kind of Array Object and can be used as the "connections" component of a Field. The shape of the grid (number of points in each dimension) are specified by a list of *n* numbers following the optional **counts** keyword in the **object** clause. The last number corresponds to the fastest varying component of the positions. The number *n* of items in this list determines the dimensionality of the grid. The last item in this list corresponds to the fastest varying dimension.

If this grid is part of a Composite Field Object, then **meshoffsets** must be specified to define where this grid is positioned relative to the entire Composite Field. The **meshoffsets** (one number for each dimension, and specified in the same order as **counts**) are the accumulated count of connections between the origin of the whole grid and the origin of this grid.

```
object         number [class]  gridconnections
               "name"
[counts]       number...
[ meshoffsets  number... ]
```

The **gridconnections** keyword does not actually correspond to a primitive Object type in the system, but instead is a convenient way of representing an important special case of the more general mesh (Mesh Array Object) of *n* 1-dimensional paths (path Array Objects). For most purposes the **gridconnections** keyword is sufficient and more convenient. The more primitive Mesh and Path Arrays are described next.

Path Array Objects

A Path Array Object encodes linear regularity of connections. It is often combined in a Mesh Array with other Path Arrays to obtain a grid of connections; in such a case, it is generally more convenient to use the **gridconnections** keyword described earlier.

A Path Array is a set of *n-1* line segments joining *n* points, where the *i*th line segment joins points *i* and *i+1*. The number of points *n* is specified by the number following the optional **count** keyword.

```
object   number [class] patharray [count] number
         "name"
```

Mesh Array Objects

A Mesh Array is a compact encoding of a generalized notion of a regular grid of connections. It is frequently used to describe a rectangular grid of connections as a product of Path Arrays; in such a case, it is generally more convenient to use the **gridconnections** keyword described earlier.

A Mesh Array encodes multidimensional regularity of connections. It is a product of connection Arrays. The product is a set of interpolation elements where the product has one interpolation element for each pair of interpolation elements in the two multiplicands, and the number of sample points in each interpolation element is the product of the number of sample points in each of the multiplicands' interpolation elements. A Mesh Array is specified by a list of **term** clauses naming the Arrays that form the product. The last term varies fastest in the resulting list of connections.

```
object   number [class] mesharray
         "name"
term     number
         "name"
         file file
         file file,number
         file file,"name"
```

Xform Objects

An xform Object specifies another Object transformed for example by a rotation, scaling, or translation. The Object to be transformed is specified by an **of** clause, and the transform itself is specified by a *3×3* matrix specified by a **times** clause and a 3-vector specified by a **plus** clause.

```
object   number [ class] xform [of]  number
         "name"                      "name"
                                     file file
                                     file file,number
                                     file file,"name"

[times]   a b c d e f g h i
[plus]   j k l
```

This Object represents the Object specified in the **of** clause, where each point *[x y z]* in the Object has been transformed to the new point *[x′ y′ z′]* according to

$$[x' \;\; y' \;\; z'] = [x \; y \; z] \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} + [j \; k \; l]$$

String Objects

A String Object encapsulates a text string as an Object. For example, the values of Object attributes are frequently string Objects. However, String Objects as such generally do not appear in Data Explorer files because a string-valued Object attribute can be specified by using the **string** keyword as described in “Objects” on page 269.

```
object   number [class] string "string" ...
         "name"
```

Light Objects

A Light Object is used to place a light in a scene for the renderer. Lights can be either local or distant. Local lights have a position and a color. Distant lights are at infinity, and have a direction and a color. It is often not necessary to specify a light in a scene because the renderer has a default built-in light that is sufficient for most purposes. For ambient lights, only **color** can be specified, not **direction** or **position**. For **distant** lights, the **direction** can be absolute, or it can be relative to the current location of the camera, as specified in a **'from camera'** clause.

```
object     number [class] light [type]   distant
           "name"                        local
                                         ambient
direction  number number number [ from camera ]
position   number number number
color      number number number
           "color"
position   number number number
```

Note: In the current release of the Data Explorer, local lights are not supported.

Camera Objects

A Camera Object specifies a camera for rendering. Camera Objects are not generally found in Data Explorer files, but rather are generated as part of the execution of a script or network. The definition of a Camera Object is included here for completeness.

```
object       number [class]
camera [[type] orthographic]
             "name"
from         number number number
to           number number number
width        number
resolution   number
aspect       number
up           number number number
angle        number
color        number number number
             color name
```

Clipped Objects

A Clipped Object represents one Object (specified by the **of** keyword) clipped by another Object (specified by the **by** keyword). Generally, Clipped Objects are not specified in input data files, but rather are generated as a result of using the ClipPlane or ClipBox module.

```
object       number [class] clipped by   number  of number
             "name"              "name"    "name"
             file file           file file
             file file,number     file file,number
             file file,"name"     file file,"name"
```

Screen Objects

Screen Objects represent an Object transformed so as always to face the camera. See Chapter 15, "Rendering" on page 149 in the *IBM Visualization Data Explorer Programmer's Reference* for more information, specifically 15.5, "Screen Class" on page 154.

```
object number [class] screen [ world    ] [ behind  ] [of] number
      "name"                 viewport   inside          "name"
                             pixel      infront         file file
                             stationary                 file file,number
                                                        file file,"name"
```

Data Mode Clause

You can specify a default data section format using the **'data mode'** clause. This clause can be used as part of an Array Object definition, or as a stand-alone clause in the header section. **msb** (most significant byte first) and **lsb** (least significant byte first) specify the byte order; **text** and **'binary'** specify the data format. (On all current platforms, **ieee** is a synonym for **binary**, and **ascii** for **text**.)

```
data mode  [  msb ] [    text ]
              lsb        ieee
                         binary
                         ascii
```

Default Clause

You can specify the default Object to be imported using the **default** clause. When a Data Explorer data file contains more than one Object (which is the usual case), the Import module (see “Import” on page 165 in *IBM Visualization Data Explorer User’s Reference*) decides which Object to import based on the value of the **variable** parameter (Object name or names) of the Import module, and the **default** clause (if specified). If **variable** is specified to Import, then those Objects specified are imported. If **variable** is not specified, but a default Object is specified with the **default** clause, then the default Object is imported. If **variable** is not specified, and no Object has been defined as the default, then the last Object in the file is imported.

```
default  number
        "name"
```

End Clause

The end of the header section is indicated by an **end** clause. The data section begins with the byte immediately following the first newline after the **end** clause.

```
end
```

B.3 CDF Files

CDF is a data abstraction for self-describing multidimensional Arrays. It represents a simpler data model than that of Data Explorer, one similar to that of the Array Object. Data are accessed in CDF through an applications programming interface, available as C and FORTRAN libraries from the National Space Science Data Center, NASA/Goddard Space Flight Center, Greenbelt, MD. Data in CDF may be stored in a number of physical formats (e.g., native or portable binary, single or multiple files, row or column majority), but the interface is the same. Hence, data in a CDF written in a format “foreign” to the workstation on which Data Explorer is running are converted automatically during the Import process.

Data Explorer provides support for importing Fields stored as CDF r-variables. To import data from a CDF, specify the CDF name as the **name** parameter in the Import Configuration dialog box (not the file name, since the CDF may be in multiple-file format). If the CDF has more than one variable, which is typical, Data Explorer categorizes each variable as positions, series, or data as appropriate. Variables that vary in one dimension only and are not record-variant are considered positions, and become the positions component in a Field Object. In many cases, these variables may have the CDF variable mnemonics of LATITUDE and LONGITUD, which are mapped to the first (x) and second (y) components of the positions vector, if they exist. This mapping permits direct use of these data with cartographic and other tools for the earth and space sciences that are publicly available for use with Data Explorer. Otherwise, the first *n* variables categorized as positions (where *n* is the dimensionality of the CDF dimensions) are used to form the positions component. Any additional such variables are treated as data variables. If there are no positions-type variables, the positions component will be a regular grid with origin of 0 and increments of 1 in each axis, where the number of axes corresponds to the dimensionality of the imported CDF r-variable.

If there are records in the CDF, each record is imported as a series member. In many cases there is a variable with the mnemonic EPOCH, which corresponds to a time stamp for each record in the CDF. If so, the double representing msec since

0 AD in each value of EPOCH is stored as the series position attribute. If not, the first variable that is record-variant and nondimensional-variant is considered the series variable. This variable is imported as the series position attribute. If there is no time variable, the series position starts at 1 and increments by 1 per series member, so that there is one member for each record in the CDF. The series position attribute, containing the time stamp, may be accessed with the Attribute module.

You can specify the name or names of the data variable in the *variable* parameter of the Import tool and the corresponding variable(s) will be imported. In the same way, you can use *start*, *end*, and *delta* to import a subset of CDF records.

Variable and global attributes present in the CDF are imported as Object attributes. These attributes may be accessed through the Attribute and Inquire modules (e.g., to build metadata-driven applications).

Variables that vary in all dimensions and are record-variant are considered data variables. Any variable that is not a position or time variable is also considered a data variable, allowing every variable to be imported. If you want the positions to be a variable other than the one chosen by Data Explorer, you can use Replace or Rename to switch the components (e.g., two or more sets of positions information are stored for different coordinate systems). Each data variable becomes a data component in a Field Object. Hence, there is one Field for each data variable in the Group imported. Since Data Explorer can handle data more flexibly than CDF, some assumptions are imposed upon certain classes of data that may be imported:

- Since data stored in CDF are not distinguished as cell-centered or node-based, all data components are treated as the latter, (i.e., data dep positions). The Post module may be used to transform a Field to cell-centered (i.e., data dep connections).
- Since CDF does not “natively” support Fields other than rank=0, all data variables are treated as scalars. The Compute module can be used to construct the appropriate vector representation from multiple scalar Fields.
- The connections component depends on the dimensionality of the data variable such that 0 = none, 1 = lines, 2 = quads, 3 = cubes, and so on.
- Each positions variable is considered a term of a Product Array to form the positions component.
- All variables of 0 dimension are imported as the data component of a Field with no positions and no connections. If the LATITUDE and/or LONGITUD variables exist, the other variables are considered data components of Fields with positions and no connections, where the positions are those latitude and longitude variables. You can construct an appropriate Field with positions and connections from the variables that are imported through modules like Construct, Regrid, and Connect.
- All variables of 1 dimension are imported as the data component of a Field of lines, where the positions would typically be a scalar (i.e., the one independent variable). If the LATITUDE and LONGITUD variables exist, then the positions are a 2-vector constructed from the latitude and longitude Arrays, but still a line.

  One-dimensional variables in CDF may be of one of three distinct classes, which are NOT distinguished in the way they are stored in a CDF file: 1) true 1-dimensional or line data; 2) indexed point data; or 3) indexed mesh data. You must know which class the variable belongs to in order to ensure that Data Explorer processes the data in an appropriate fashion. The first class is handled correctly. For the second and third class, the connections component of any imported Field(s) may be meaningless. You can use the Remove module to eliminate it and treat the Field as scattered or point data (i.e., use Regrid or Connect to create a more appropriate mesh).

Treating such data as a collection of points is consistent with the original design philosophy of CDF and CDF applications. The third case actually represents an irregular mesh, which Data Explorer can support directly. Unfortunately, the connectivity information (i.e., the mesh structure) is typically not stored in the CDF, so Import cannot directly reconstruct the original mesh. Hence, the data must be treated as point data unless you have information, external to the CDF, that can be used to recreate the original mesh structure.

B.4 netCDF Files

Data Explorer supports the importation of data in netCDF format, a data abstraction for self-describing multidimensional Arrays. It represents a simpler data model than that of Data Explorer, one similar to that of the Array Object. Data are accessed in netCDF through an application programming interface (available in C and FORTRAN libraries from the Unidata Program Center—in Boulder, Colorado).

Scalar data on a regular grid can be imported from a standard netCDF file. To import vector data, data on irregular grids, or time series data, additional attributes must be added to the netCDF file. These attributes allow you to specify the data, positions, and connections components of your data set. See B.5, "netCDF Files: Complex Fields" on page 282 for more information about these attributes.

Regular Grids

To import scalar data on a regular grid, specify the netCDF file name as the **name** parameter. By default, all netCDF variables are imported and collected into a Group. To import one or more particular variables, specify their names as the **variable** parameter. The **format** parameter must be "netCDF."

Data Explorer automatically constructs positions and connections for each variable, with an origin of 0.0 and spacings of 1.0 along each dimension.

For data that is logically a vector Field, but whose values are stored in three separate netCDF variables, each component of the vector can be imported separately; the Compute module can then be used to create a single vector Field.

For data that is logically a vector Field, but whose values are stored as an *n*+1 dimensional regular grid, use the Slice and Compute modules to separate the components of the vector, and then recombine them into a single vector Field.

Example of a Regular Grid

The following file describes a 3 × 3 × 3 regular grid at origin 0, 0, 0 with deltas of 1.0 along each axis.

```
netCDF volume {

dimensions:
     nx = 3;
     ny = 3;
     nz = 3;
variables:
          float field_data(nx, ny, nz);

data:
          field_data =
         0, 0, 0
         0, 0, 0
         0, 5, 0

          0, 0, 5
          0, 0, 0
          0, 0, 0


          5, 0, 0
          0, 0, 0
          0, 0, 0;
}
```

netCDF on completely regular grids can be imported directly by Data Explorer without modifying the netCDF file. See B.4, "netCDF Files" on page 281 for more information.

B.5 netCDF Files: Complex Fields

For data with more complex structure, conventions have been established for netCDF variable attributes, as described in the format below. The notation used corresponds to that of the netCDL "language."

Irregular Arrays

This section describes how to specify netCDF variables for components with irregular values.

Data

To indicate that a netCDF variable contains values corresponding to the data component, it must have the following attribute:

```
          variable1:field = "fieldname";
```

Variable1 is the name of the netCDF variable containing data values to be imported. *fieldname* is the name of the Data Explorer field by which the user refers to the data (for example, "temperature," "pressure," "wind"). If more than one variable is tagged with the same field name, each variable is read into a field, and the fields are collected into a group.

The data are read in as an array of values, one number per grid point. If the data are actually a vector or a matrix at each grid point, use one of the following modifiers:

- `variable1:field = "fieldname, vector";`
- `variable1:field = "fieldname, matrix";`

The non-scalar data are stored in additional dimensions for the variable. For a static three-dimensional 3-vector, the three components are stored in a fourth dimension of size 3.

If the data have both regular connections and regular positions, no other attributes are required. A regular grid is assumed, with the origin at 0.0, and a spacing of 1.0 along each axis. The number of axes will be determined from the number of dimensions in the data array.

Positions

If the locations of the data values in *variable1* do not form a regular lattice (with origins at 0.0 and spacings of 1.0), the name of a netCDF variable that contains the position information must be specified as an attribute for *variable1*.

There are five different types of position specifications: none, completely regular, completely irregular, and two types of partially regular.

Completely irregular is assumed if the following attribute is specified:

```
variable1:positions = "variable2";
```

where *variable2* is an array of vectors, one for each grid point, defining its location. The dimensionality of the data space is determined by the number of items in a vector.

Regular positions can be specified with just the origin and spacing between grid points along each axis in compact form. The following attribute is used:

```
variable1:positions = "variable2, compact";
```

where *variable2* is the name of a *n*×*2* array containing origin, delta pairs for the spacing and location of positions along each axis. The number of positions along each axis is determined from the shape of *variable1*.

Positions that can be specified as the product of arrays containing the location of points along each axis can be input in product form. Use the following attribute:

```
variable1:positions = "variable2a, product;
                                  variable2b, product;
                                        .
                                        .
                                        .
                                  variable2x, product";
```

where the *variable2*'s are each the name of an array containing a list of positions along that axis. The number of items in each array must match the length of the corresponding axis in the original *variable1* data array.

If any of the axes in an partially regular product array are actually regular, they can be specified in `compact` form:

```
variable1:positions = "variable2a, product, compact;
                                  variable2b, product;
                                    .
                                    .
                                    .
                                  variable2x, product";
```

where *variable2a* is the name of an origin, delta array, and the rest are position lists as before.

Connections

If the connections between positions is a regular lattice, no additional attributes are necessary. For 1-D data, connections of “lines” is assumed. 2-D data implies “quads,” 3-D data implies “cubes” and for higher dimensions, “hypercubes” is assumed.

If the connections are irregular, use one of the following attributes:

- `variable1:connections = "variable3, tetrahedra";`
- `variable1:connections = "variable3, triangles";`
- `variable1:connections = "variable3, cubes";`
- `variable1:connections = "variable3, quads";`

where *variable3* is the name of an array containing a vector of point numbers, defining each connection element item. The length of this vector depends on the choice of connections. If the shape is not explicitly specified, *tetrahedra* are assumed.

Additional Components

If additional component information is present in the file, the following attributes are valid:

```
variable1:component = "variable4, componentname, scalar;
                       variable5, componentname, vector;
                       variable6, componentname, matrix";
```

and

```
variable4:attributes = "ref, componentname;
                        dep, componentname";
```

Series Data

There are three ways to specify the import of datasets that should be treated as series. They are:

- Single variable
- Separate variables
- Separate files

Single Variable

When all data values are defined as a single netCDF variable, and the unlimited dimension of the variable is to be interpreted as the series dimension, then use one of the following forms of the `field` attribute:

- `variable1:field = "fieldname, scalar, series";`
- `variable1:field = "fieldname, vector, series";`
- `variable1:field = "fieldname, matrix, series";`

All other specifications are the same as for simple fields.

The position and connection information is assumed to be constant for all members of the series. If the positions or connections change for each step of the series, then the variables used for those arrays must also have an unlimited dimension that corresponds one-for-one with the data array.

An example using this method is provided in “Partially Regular Grids and Time Series” on page 286.

Separate Variables

When there are separate netCDF variables defined for each step in the series, but all variables are in the same file, use the following global attribute tags:

```
    :seriesxxx = "fieldname;
                  variable1a;
                  variable1b;
                         .
                         .
                         .
                  variable1x";
```

or

```
    :seriesxxx = "fieldname;
                  variable1a, float_value;
                  variable1b, float_value;
                         .
                  variable1x, float_value";
```

where the global tag must have the first 6 characters `series`. Global tags must be unique, so additional characters can be added to distinguish them.

Each *variable1*x is the name array containing the data for that step. In the first format, the spacing of the steps is assumed to be 1.0. In the second format, the *float_value* is the value of each step. All other specifications are the same as for simple fields.

Separate Files

When there are netCDF variables in separate files that make up the steps of a series, use the following global attribute tags:

```
    :seriesxxx = "fieldname, files;
                  filename1;
                  filename2;
                         .
                         .
                         .
                  filenameN";
```

or

```
    :seriesxxx = "fieldname, files;
                  filename1, float_value;
                  filename2, float_value;
                         .
                         .
                         .
                  filenameN, float_value";
```

where the global tag must have the first 6 characters `series`. Global tags must be unique, so additional characters can be added to distinguish them.

Each *filename*N is the name of the netCDF file that contains the data variables for that step. In the first format, the spacing of the steps is 1.0. In the second format, the *float_value* is the value of each step. All other specifications are the same as for simple fields.

This format can be used to create short term series within a file, and then have a series of these smaller series.

Examples

This section shows examples of netCDF files in the netCDL description language. See the documentation supplied by UCAR for more information on netCDL and the **ncgen** and **ncdump** utilities.

Compact Specifications of Regular Dimensions

This example describes a single two-dimensional scalar field on a latitude-longitude, regular, rectangular grid. The example data are temperature on a one-degree grid with global coverage. Because Data Explorer array objects can be specified compactly, you can use this method to specify a netCDF with regular dimensions. For each dimension, you need to specify its value at the origin and its spacing along the dimension.

In this example, two variable attributes are defined for the netCDF variables. **field** specifies the rank of the parameter, and **positions** specifies where the information containing the locations of the data is space is located.

```
dimensions:
    lon = 360;
    lat = 180;
    naxes = 2;
    ndeltas = 2;

variables:
    float locations(naxes, ndeltas);
    float temperature(lat, lon);
    temperature:field = "temperature, scalar";
    temperature:positions = "locations, regular";

data:
    locations = 89.5, -1.,      // compact specification, origin and
                -179, 1.;       // spacing for lat and lon
    temperature = ...       // Data for temperature
```

Partially Regular Grids and Time Series

This example describes an ocean circulation model that consists of a time series of four three-dimensional scalars **(temp**, **sali**, **wata**, and **conv)** and one three-dimensional 3-vector **(vel)**. netCDF typically requires seven variables, all scalars (the vector counting as three scalars). The coordinate system for the velocity vectors corresponds to that of the grid (that is, +u implies north, +v implies east, and +w implies down).

These grids are partially regular in that the **time**, **tlat**, and **tlon** portions (three out of the four dimensions) are all regularly spaced. **time** is to be mapped to members of a series group. The fourth dimension, **tlvl**, is irregularly spaced. The compact notation can be used for the regular notation, while the all values along the irregular dimension must be specified; a product is formed from the dimensions.

Here is the specification in netCDL notation:

```
dimensions:
    time = UNLIMITED;
    tlat = 30;
    tlon = 50;
    tlvl = 30;
    vsize = 3;   // At each grid cell for variable vel, there are
                 // three floats for the u, v, and w components of the
                 // vector field.
    naxes = 3;
    ndeltas = 2;

variables:
    float lat_axis(ndeltas, naxes);
    float lon_axis(ndeltas, naxes);
    float level_axis(tlvl, naxes);

    float temp(time, tlat, tlon, tlvl);
    temp:field = "temperature, scalar, series";
    temp:positions = "lat_axis, product, compact; lon_axis,
                      product, compact; level_axis, product";

    float sali(time, tlat, tlon, tlvl);
    sali:field = "salinity, scalar, series";
    sali:positions = "lat_axis, product, compact; lon_axis,
                      product, compact; level_axis, product";

    float wata(time, tlat, tlon, tlvl);
    wata:field = "water parage, scalar, series";
    wata:positions = "lat_axis, product, compact; lon_axis,
                      product, compact; level_axis, product";

    float conv(time, tlat, tlon, tlvl);
    conv:field = "covective index, scalar, series";
    conv:positions = "lat_axis, product, compact; lon_axis,
                      product, compact; level_axis, product";

    float vel(time, tlat, tlon, tlvl, vsize);
    vel:field = "velocity, vector, series";
    vel:positions = "lat_axis, product, compact; lon_axis,
                     product, compact; level_axis, product";

data:
    lat_axis = -14.667, 0., 0.,
                 0.333, 0., 0.;
    lon_axis = 0.0, -99.8, 0.0,
               0.0, 0.5, 0.0;
    level_axis = 0.0, 0.0, 17.5,
                 0.0, 0.0, 53.425,
                 .
                 :
                 0.0, 0.0, 5374.98;
    temp = ... ;
    sali = ... ;
    wata = ... ;
    conv = ... ;
    vel = ... ;
```

Irregular Surface

This example is the netCDL description of a netCDF for an irregular surface, that of the classic teapot. It has precomputed normals, which are imported as the "normals" component, in addition to positions and connections.

```
netcdf teapot8 {    // name of datafile is "teapot8.ncdf"
                    // name of field is "surface"

dimensions:
        pointnums = 2268;
        trinums = 3584;
        axes = 3;
        sides = 3;

variables:
        float locations(pointnums, axes);
        float normalvect(pointnums, axes);
        long tris(trinums, sides);
        float surfacedata(pointnums);

// global attributes:
                :source = "Classic Teapot, data from Turner Whitted";

// specific attributes:
                surfacedata:field = "surface";
                surfacedata:connections = "tris, triangles";
                surfacedata:positions = "locations";
                surfacedata:component = "normalvect, normals, vector";

                normalvect:attributes = "dep, positions";

// This is the start of a large data section
data:
⋮
}
```

B.6 HDF Files

HDF is a multiobject file structure that is designed to facilitate the transfer of data between machines. HDF was created at the National Center for Supercomputing Applications (NCSA).

Data Explorer provides support for importing HDF files that contain a Scientific DataSet (SDS). A Scientific DataSet is an HDF set that stores rectangular gridded Arrays of data, together with the information about the data.

Note: Scientific Data Sets should be created using the DFSD API and not the SD API.

To import HDF files, specify the filename as the **name** parameter. By default, all the datasets will be imported and collected into a Group. To import a particular dataset specify a number corresponding to that dataset as the **variable** parameter (0 corresponds to the first dataset). The **format** parameter must be "hdf".

If dimension scales are specified, Data Explorer uses these to construct positions; otherwise positions have an origin of 0.0 and deltas of 1.0 along each dimension. Data Explorer automatically constructs regular connections for either case.

Appendix C. Environment Variables and Command Line Options

C.1 Environment Variables

The environment variables described in this section can be set in your login profile to customize Data Explorer. Note also that these variables can be overridden on the command line (see C.2, “Command Line Options” on page 295).

Path Variables

Path variables specify a directory or directories to be searched for files. Directories are searched in the order of their appearance in the variable, reading from left to right, with successive path names separated by a colon (:). Thus when a file appears in more than one directory, Data Explorer will choose the first copy it finds (i.e., in the leftmost directory containing a copy).

DXDATA specifies directories to be searched for importable data files. If the data to be imported is in your current directory or one of the specified directories, you do not need to enter the complete path name in the Configuration dialog box for the Import tool: given just the file name, the Import module will search all of these directories.

DXINCLUDE specifies directories to be searched for include scripts. Data Explorer uses include scripts in script mode. It is not necessary to specify this variable in Edit mode.***:*** See “File Inclusion” on page 207 for more information.

DXMACROS specifies directories to be searched for macros when Data Explorer starts up. If DXMACROS is not specified, you will have to load macros individually (see 7.2, “Creating and Using Macros” on page 149).

DXMODULES specifies the directories to be searched for outboard modules.

Setting a Path Variable: Examples

Note the colon (:) separating successive path names.

- To set DXMACROS for both the Bourne (sh) and the Korn (ksh) shells:

```
DXMACROS=/usr/mydirectory/projectAmacros:/usr/mydirectory/projectBmacros
export DXMACROS
```

- To set DXDATA for the Korn shell (ksh) only:

```
export DXDATA=/usr/mydirectory/mydata:/usr/group/groupdata
```

- To set DXDATA for the C shell (csh):

```
setenv DXDATA /usr/mydirectory/mydata:/usr/group/groupdata
```

Other Environment Variables

DX8BITCMAP sets the level at which the change to using a private color map is made. The allowed values are -1 and the range from 0 (zero) to 1 (one) and represent the Euclidean distance in RGB color space, normalized to 1 (one) for the maximum allowed discrepancy. The default value is 0.1. If this variable is set to 1, a private color map will never be used; conversely, if it is set to -1, a private color map will always be used. (See Display in *IBM Visualization Data Explorer User’s Reference*.)

DXARGS specifies the default set of arguments for Data Explorer start-up. An option specified on the command line will override the corresponding setting in the variable.

DXAXESMAXWIDTH sets the number of digits in axes tick labels at which a switch to scientific notation is made. The default is 7.

DXNO_BACKING_STORE if set to anything, disables framebuffer readbacks. Setting this environment variable will improve performance of interaction with hardware rendered images, especially for machines for which readback is slow. However, some of the interactions in the image window (such as zoom) will result in a black image while interaction is taking place. If you are not planning on using the Image tool, then it is strongly recommended that this environment variable be set. The default is that framebuffer readbacks are enabled.

DXCOLORS specifies a file name containing string and RGB value pairs as an alternate for /usr/lpp/dx/lib/colors.txt. The string name can be used by any Data Explorer tool where a color can be specified by name (for example, Color). The RGB value specifies the specific numeric value for the color.

DXDELAYEDCOLORS enables ReadImage to create delayed color images if the image is a tiff format image saved in a byte-with-colormap format or a GIF format. This feature is enabled if this variable is set to any value. Delayed colors use less memory.

DXEXEC specifies an executive to be run at start-up. You should set this variable only for a customized version of Data Explorer.

DXFLING If DXFLING is set to 1, then for hardware-rendered images, in rotation mode and execute-on-change mode, if you drag the mouse across the image, and release the mouse button outside the image, the object in the image will begin to rotate, and will continue to rotate until you click inside the image. The direction and speed of the mouse motion before release will affect the rotation direction and rotation speed of the object in the window.

DXGAMMA sets the gamma correction for software-rendered images displayed to the screen by a Display or Image tool. On many display devices a given change in the digital brightness of the image is not reflected in a corresponding change in screen brightness. A gamma correction is a nonlinear adjustment of the pixel values to compensate for this difference and produce a more accurate representation on the screen. By default (except for 8-bit windows on the sgi architecture), the correction factor (exponent) is 2 (two), on the assumption that the display is not otherwise gamma corrected. The `DXGAMMA` variable allows you to override this default. In particular, if the display device is already gamma corrected, set the variable to 1 (one). (See Display in *IBM Visualization Data Explorer User's Reference*, and README_sgi in /usr/lpp/dx.)

DXGAMMA_8BIT, DXGAMMA_12BIT, and DXGAMMA_24BIT set the gamma correction for software-rendered images displayed to the screen in 8-, 12-, or 24-bit windows by a Display or Image tool. This variable overrides the value set by DXGAMMA.

DXHOST specifies the machine name of the server on which the executive is to be run. The default is “localhost”. (See 9.3, “Connecting to the Server” on page 183 for information on how to connect to the server.) To determine the host name, enter the command:

```
uname -n
```

DXHWGAMMA sets the gamma correction for hardware-rendered images displayed to the screen by a Display or Image tool. On many display devices a given change in the digital brightness of the image is not reflected in a corresponding change in screen brightness. A gamma correction is a non-linear adjustment of the pixel values to compensate for this difference and produce a more accurate representation on the screen. By default, the correction factor is 2, on the assumption that the display is not otherwise gamma corrected. The `DXHWGAMMA` variable allows you to override this default. In particular, if the display device is already gamma corrected, set the variable to 1.

DXHWMOD if both GL and OpenGL are supported, you can override the default library (which is platform-specific; please see the appropriate README file for your architecture in `/usr/lpp/dx`) by using this environment variable. It should be set to either DXhwdd.o (for GL) or DXhwddOGL.o (for OpenGL).

DXMDF specifies the name of the .mdf file that contains custom-added modules for customized versions of Data Explorer.

DXMEMORY sets the amount of memory (in megabytes) that can be used by the executive.

DX_NESTED_LOOPS for faces, loops, and edges data, if set, allows loops other than the enclosing loop for a face to be listed first. However, there is a consequent decrease in performance if this environment variable is set.

DXPIXELTYPE sets the image type to either 24-bit color images or floating-point-based 96-bit images (the default). This affects the behavior of Render and ReadImage. This variable can be set to either DXByte (24 bits) or DXFloat (96 bits). Setting this variable to DXByte will result in images taking up less memory.

DXPROCESSORS sets the number of processors for Data Explorer SMP.

DXROOT specifies the top-level directory for all the files and directories needed by Data Explorer. The default is /usr/lpp/dx.

DXSHMEM specifies whether or not shared memory should be used. The amount of memory allocated by Data Explorer for its data and object management can be set at runtime with the `-memory` command line option. At startup, Data Explorer either allocates a shared memory segment or expands the existing data segment to create this space.***:*** SMP (multiprocessor) systems are required to use shared memory so each processor can share a common data space. SGI systems also use shared memory for space. IBM systems use shared memory if the size to be allocated is larger than 256 MB. In all other cases Data Explorer extends the existing data segment using the brk() system call.

Each architecture (SGI, IBM, HP, ...) has a different way of configuring the maximum user data segment size, and a different way of setting the limit on the maximum size of a single shared memory segment. Consult your system administrator or system documentation if you have problems getting Data Explorer to use the amount of memory which should be available to you.

If you have problems using a large data segment, you can force Data Explorer to use shared memory by setting the `DXSHMEM` environment variable to any value other than -1. This will override the defaults and use shared memory for space.

Alternatively, you can force Data Explorer to extend the data segment (if allowed for the architecture) by setting DXSHMEM to -1.

Note: Regardless of the setting of DXSHMEM, the aviion and sun4 architectures always use the data segment.

DXSHMEMSEGMAX Some architectures have a default configuration which limits the size of shared memory segments (see the architecture specific README file in /usr/lpp/dx), and the system configuration must be changed as root to increase the maximum allowed size of a shared memory segment. If the maximum is not reset or if it is already set to a different limit, then you can use DXSHMEMSEGMAX to tell Data Explorer what the current limit is in megabytes (e.g. 128 == 128 MB). Data Explorer will allocate multiple shared memory segments if necessary to get the total amount of space, but it must be able to allocate them at contiguous virtual memory addresses.

DXTRIALKEY can be used in place of the expiration file ($DXROOT /expiration) for a trial license. The value of the variable is the string specifying the trial key. It takes precedence over $DXROOT/expiration and DXTRIALKEYFILE.

DXTRIALKEYFILE specifies the name of the expiration file for a trial license. It takes precedence over $DXROOT/expiration.

DX_USER_INTERACTOR_FILE Specifies a file containing user interactors for use by the SuperviseState and SuperviseWindow modules (see “SuperviseState” on page 332 and “SuperviseWindow” on page 336 in *IBM Visualization Data Explorer User’s Reference*).

C.2 Command Line Options

Table 5 lists the command line options available with Data Explorer. Those most commonly used are identified by a bullet (●). Table 6 on page 297 lists command line options of particular interest to developers.

Command line options always override corresponding environment variables. As such, they offer a quick way to temporarily override environment settings. If parameters conflict, the last one entered takes precedence.

Table 5 (Page 1 of 3). Data Explorer Command Line Options

	Option syntax	**Function**
	-8bitcmap [*private*|*shared*|*0–1*]	Set color-map error threshold (default: *0.1*). private = -1; stored = 1.
●	-builder	Start the Data Explorer Module Builder (instead of Data Explorer).
	-cache [*on*|*off*]	Enable executive cache (default: *on*).
	-colors *filename*	Override DXCOLORS environment variable.
	-connect *host:port*	Start a distributed executive only (no user interface).
●	-data *pathlist.*	Override DXDATA environment variable.
	-directory *dirname*	Change directory (cd) to *dirname* before starting the executive.
	-display *hostname:0*	Set the X-display destination.

Table 5 (Page 2 of 3). Data Explorer Command Line Options

	Option syntax	**Function**
	-dxroot *dirname*	Set the Data Explorer root directory (default: */usr/lpp/dx*).
	-echo	Echo the command lines without executing them.
	-edit	Start the user interface in edit mode (default).
	-exec *filename*	Use the specified executive.
	-execonly	Start the executive only (no user interface) in remote mode.
	-execute	Execute the visual program automatically at start-up.
	-execute_on_change	Go into Execute On Change mode at start-up.
	-full	Start the full Data Prompter. (See also -file and -prompter.)
	-file *filename*	Start the Data Prompter with header file *filename*. (See also -full and -prompter.)
	-help	Print the abbreviated help message.
	-highlight [*on*|*off*]	Enable node execution highlighting: (default: *on*).
●	-host *hostname*	Start the executive on machine. *hostname*.
●	-hwrender [gl | opengl]	if both GL and OpenGL are supported, set the type of hardware-rendering used.
●	-image	Start the user interface in image mode.
	-include *pathlist*	Override DXINCLUDE environment variable.
	-license [*runtime*|*develop*|*timed*]	Request that the user interface use only the indicated functional type
	-local	Start the executive on the current machine (default).
	-log [*on*|*off*]	Enable executive and user interface logging (default: *off*).
●	-macros *pathlist*	Set list of directories to be searched for macros.
	-mdf *filename*	Use .mdf file *filename* in addition to the default mdf file.
●	-memory *#Mbytes*	Set the amount of memory the executive uses.
●	-menubar	Start the user interface in menubar mode.
	-metric	Set the graphical user interface to use metric units whenever possible.
	-modules *pathlist*	Set list of directories to be searched for outboard modules.
●	-morehelp	Print complete Help, including information about other options.
●	-optimize [*memory*|*precision*]	Set the environment variables DXPIXELTYPE and DXDELAYEDCOLORS to optimize memory or precision. (The default is precision.)
	-outboarddebug	Enable user to start outboard modules manually.
●	-program *filename*	Start the user interface with visual program *filename*.
●	-prompter	Start the Data Explorer Data Prompter (but not Data Explorer).

Table 5 (Page 3 of 3). Data Explorer Command Line Options

	Option syntax	Function
	-readahead [*on*|*off*]	Enable executive readahead: (default: *on*).
●	-script	Run the executive only (i.e., in script mode).
●	-script *filename*	Run the executive only (i.e., in script mode) with script *filename*.
	-suppress	Do not open any control panels at start-up (in image or menubar mode only).
	-timed	Start Data Explorer using a timed license.
	-timing [*on*|*off*]	Enable module timing (default: *off*).
	-trace [*on*|*off*]	Enable executive trace (default: *off*).
	-trialkey	Automatically determines the information needed to generate a trial key.
●	-tutor	Start the Data Explorer tutorial.
●	-uionly	Start the user interface only (no executive).
	-verbose	Echo command lines before executing them.
	-version	Show version numbers of dxexec and dxui.

Table 6 (Page 1 of 2). Command Line Options for Developers. For more detailed descriptions of functions, see Appendix D, "User Interface Configuration" on page 299.

Option syntax	Function
-encode -key *<16-digit hex number> <file>*	Encode a .net file. The Visual Program Editor will not display an encoded .net file, and such a file cannot be saved. (However, .cfg configuration files *can* be saved.)
-key *<16-digit hex number> <file>*	Decodes an encoded .net file.
-limitImageOptions	Remove options from Image window's Options menu.
-noAnchorAtStartup	Start Data Explorer, but do not put up any windows by default
-noCMapOpenMap	Remove Open... option from Colormap Editor's File menu.
-noCMapSaveMap	Remove Save As... option from Colormap Editor's File menu.
-noCMapSetNameOption	Remove Set Colormap Name... option from Colormap Editor's Options menu.
-noConnectionMenus	Remove Connection menu from all Windows (intended for use with DXLink applications).
-noConfirmedQuit	Turn off the "Are you sure you want to quit Data Explorer?" message.
-noDXHelp	Remove three options from the Help menu of the control panel and the Image window.
-noEditorAccess	Remove options from the Edit menu of the control panel and the Open Visual Program Editor option from the Image window's Windows menu.

Table 6 (Page 2 of 2). Command Line Options for Developers. For more detailed descriptions of functions, see Appendix D, “User Interface Configuration” on page 299.

Option syntax	**Function**
-noEditorOnError	Turn off default behavior of popping up a VPE when an error occurs. Instead, a dialog box will ask whether a VPE should be opened.
-noExecuteMenus	Remove Execute menu from all Windows (intended for use with DXLink applications).
-noExitOptions	Changes“quit” to“Close” (intended for use with DXLink applications).
-noImageLoad	Remove both Load options from the Image window's File menu.
-noImageMenus	Remove menus from all Image and Display windows.
-noImagePrinting	Remove the Print Image... option from the Image Window's File menu.
-noImageRWNetFile	Remove Open, Save, and Save As... options from the Image Window's File menu.
-noImageSaving	Remove Save Image... option from the Image Window's File menu.
-noInteractorAttributes	Remove Set Attributes... option from the Image Window's File menu.
-noInteractorEdits	Remove four set options from the control panel's Edit menu.
-noInteractorMovement	Restricts the ability to move interactor instances within a control panel.
-noMessageInfoOption	Remove the Information Messages toggle button from the Message window’s Options menu.
-noMessageWarningOption	Remove the Warning Messages toggle button from the Message window's Options menu.
-noOpenAllPanels	Remove Open All Panels... option from the Panels menu of the control panel and from the Windows menu of the Image Window.
-noPanelAccess	Remove the Panels menu from the control panel and panel options from the Windows menu of the Image Window.
-noPanelEdit	Remove the Edit menu from the control panel.
-noPanelOptions	Remove the Option menu from the control panel.
-noPGroupAssignment	Remove the Execution Group Assignment... option from the Image window's Connection menu.
-noRWConfig	Remove the Open... and Save As... options from the control panel's File menu.
-noScriptCommand	Remove the Execute Script Command... option from the Message Window's Options menu.
-restrictionLevel	Combines options to facilitate building applications for Data Explorer

Appendix D. User Interface Configuration

Data Explorer provides its own set of resources for customizing the user interface in addition to the standard X-window and Motif resources. These resources (see table) can be specified in the `.Xdefaults` resource file, and a majority can be invoked as command line switches or options. For example:

```
DX*noImagePrinting: true
DX*restrictionLevel: maximum
```

are valid resource-file entries. The corresponding command line format is:

```
dx -image -noImagePrinting -restrictionLevel maximum
```

The command line switch for a Boolean resource (e.g., `-noImagePrinting`) toggles the default value.

Table 7 (Page 1 of 2). Resource Configuration Table

Resource Name	Command Line Option (if available)	Type	Default
DX*errorEnabled	N/A	Boolean	true
DX*errorOpensMessage	N/A	Boolean	true
DX*infoEnabled	N/A	Boolean	true
DX*infoOpensMessage	N/A	Boolean	true
DX*limitImageOptions	-limitImageOptions	Boolean	false
DX*metric	-metric	Boolean	false
DX*noConfirmedQuit	-noConfirmedQuit	Boolean	false
DX*noCMapOpenMap	-noCMapOpenMap	Boolean	false
DX*noCMapSaveMap	-noCMapSaveMap	Boolean	false
DX*noCMapSetNameOption	-noCMapSetNameOption	Boolean	false
DX*noDXHelp	-noDXHelp	Boolean	false
DX*noEditorAccess	-noEditorAccess	Boolean	false
DX*noEditorOnError	-noEditorOnError	Boolean	false
DX*noImageLoad	-noImageLoad	Boolean	false
DX*noImageMenus	-noImageMenus	Boolean	false
DX*noImagePrinting	-noImagePrinting	Boolean	false
DX*noImageRWNetFile	-noImageRWNetFile	Boolean	false
DX*noImageSaving	-noImageSaving	Boolean	false
DX*noInteractorAttribute	-noInteractorAttribute	Boolean	false
DX*noInteractorEdits	-noInteractorEdits	Boolean	false
DX*noInteractorMovement	-noInteractorMovement	Boolean	false
DX*noMessageInfoOption	-noMessageInfoOption	Boolean	false
DX*noMessageWarningOption	-noMessageWarningOption	Boolean	false
DX*noOpenAllPanels	-noOpenAllPanels	Boolean	false
DX*noPGroupAssignment	-noPGroupAssignment	Boolean	false
DX*noPanelAccess	-noPanelAccess	Boolean	false
DX*noPanelEdit	-noPanelEdit	Boolean	false
DX*noPanelRWConfig	-noPanelRWConfig	Boolean	false
DX*noScriptCommand	`-noScriptCommand`	Boolean	false

Table 7 (Page 2 of 2). Resource Configuration Table

Resource Name	Command Line Option (if available)	Type	Default
DX*printImageCommand	N/A	String	“lpr”
DX*printImageFormat	N/A	String	“PSCOLOR”
DX*printImagePageSize	N/A	String	NULL
DX*printImageResolution	N/A	int	0
DX*printImageSize	N/A	String	NULL
DX*restrictionLevel	-restrictionlevel	String	NULL
DX*saveImageFormat	N/A	String	“PSCOLOR”
DX*saveImagePageSize	N/A	String	NULL
DX*saveImageResolution	N/A	int	0
DX*saveImageSize	N/A	String	NULL
DX*vpeCanvas.lineThickness	N/A	int	1
DX*warningEnabled	N/A	Boolean	true
DX*warningOpensMessage	N/A	Boolean	true

DX*errorEnabled

Specifies the default value of the **Error Messages** toggle button in the Message window’s **Option** menu.

DX*errorOpensMessage

Specifies whether or not the Message window pops up when an error message is printed in the Message window.

DX*infoEnabled

Specifies the default value of the **Information Messages** toggle button in the Message window’s **Option** menu.

DX*infoOpensMessage

Specifies whether or not the Message window pops up when an informational message is printed in the Message window.

DX*limitImageOptions

Removes the **Image Depth**, **Throttle**, **Change Image Name...**, and **Panel Access...** menu commands from the Image window’s **Options** menu.

DX*metric

Specifies that Data Explorer use the metric system (centimeters) when set to “true”. The default is the English system (inches). The **Print Image** and **Save Image** dialog boxes option menus are affected accordingly.

DX*noConfirmedQuit

Turns off the “Are you sure you want to quit Data Explorer?” message.

DX*noCMapOpenMap

Removes the **Open...** menu command from the Colormap Editor’s **File** menu.

DX*noCMapSaveMap

Removes the **Save As...** menu command from the Colormap Editor’s **File** menu.

DX*noCMapSetNameOption

Removes the **Set Colormap Name...** menu command from the Colormap Editor’s **Options** menu.

DX*noDXHelp

Removes the **On Context**, **On Manual** and **On Help** menu commands from the Help menu of both the Control Panel and the Image window.

DX*noEditorAccess

Removes (1) the **Delete**, **Show Selected Interactor**, **Add Selected Interactor**, **Show Selected Tool**, and **Comment...** menu commands from the Control Panel’s **Edit** menu; (2) the **Open Visual Program Editor** menu commands from the Image window’s **Windows** menu.

DX*noEditorOnError

Turns off default behavior of popping up the VPE when an error occurs. Instead, a question dialog box is popped up, with the name of the visual program, which lets the user choose whether or not to open the VPE on the .net file.

DX*noImageLoad

Removes the **Load Macro...** and **Load Module Definition(s)...** menu commands from the Image window’s **File** menu.

DX*noImageMenus

Removes the bar from all Image and Display windows.

DX*noImagePrinting

Removes the **Print Image...** menu command from the Image window’s **File** menu.

DX*noImageRWNetFile

Removes the **Open**, **Save**, and **Save As...** menu commands from the Image window's **File** menu.

DX*noImageSaving

Removes the **Save Image...** menu command from the Image window's **File** menu.

DX*noInteractorAttribute

Removes the **Set Attributes...** menu command from the Control Panel's **Edit** menu.

DX*noInteractorEdits

Removes the **Set Style**, **Set Layout**, **Set Dimensionality**, and **Set Interactor Label...** menu commands from the Control Panel's **Edit** menu.

DX*noInteractorMovement

Restricts the ability to move interactor instances within a Control Panel.

DX*noMessageInfoOption

Removes the **Information Messages** toggle button from the Message window's **Options** menu.

DX*noMessageWarningOption

Removes the **Warning Messages** toggle button from the Message window's **Options** menu.

DX*noOpenAllPanels

Removes the **Open All Panels...** menu command from the **Panels** menu of the Control Panel and the **Windows** menu of the Image window.

DX*noPGroupAssignment

Removes the **Execution Group Assignment...** menu command from the Image window's **Connection** menu.

DX*noPanelAccess

Removes (1) the Control Panel's **Panels** menu and all its menu commands; (2) the **Open All Control Panels** and **Open Control Panel by Name** menu commands from the Image window's **Windows** menu.

DX*noPanelEdit

Removes the Control Panel's **Edit** menu and all its menu commands. This is equivalent to the following options:

```
noInteractorEdits
noInteractorAttribute
noEditorAccess
noInteractorMovement
```

By restricting the ability to highlight interactors with a mouse click, this option in effect also disables interactor movement.

DX*noRWConfig

Removes the **Open...** and **Save As...** menu commands from the **File** menu.

DX*noScriptCommand

Removes the **Execute Script Command...** menu command from the Message window's **Options** menu.

DX*printImageCommand

Specifies the default print image command to be used use in the **Print Image** dialog box available from the Image window.

DX*printImageFormat

Specifies the default file format to be used in the **Print Image** dialog box available in the Image window. The following values are recognized:

PSCOLOR	Color PostScript
PSGREY	Gray level PostScript
EPSCOLOR	Encapsulated Color PostScript
EPSGREY	Encapsulated Gray level PostScript

DX*printImagePageSize

Specifies the default page size that PostScript images should be centered on when printing images from the **Print Image** dialog box available from the Image window. Units are those indicated by the metric option.

DX*printImageResolution

Specifies the default resolution that PostScript images should be printed with when printing images from the **Print Image** dialog box available from the Image window. Resolution is in dots per inch unless the `DX*metric` resource is set, in which case dots per centimeter is used. This resource is overridden by the `printImageSize` resource. If this is set to zero (the default), then the **Print Image...** dialog box chooses the resolution.

DX*printImageSize

Specifies the default size of PostScript images printed from the **Print Image** dialog box available from the Image window. If specified this option overrides the `printImageResolution` option. The value is a string in the same format as that accepted by the dialog box (for example, “8”, “8x”, “x10”, “8x10”). Units are those indicated by the metric option. The default value is determined by `printImageResolution` option.

DX*restrictionLevel

Combines options to make it easier to build applications on top of Data Explorer. The string value must be one of the following:

minimum	Combines the restrictions implied by the following options:
	`noEditorAccess`
	`noInteractorEdits`
	`noInteractorMovement`
	`limitImageOptions`
	`noScriptCommand`
	`noCMapSetNameOption`

intermediate	Combines the restrictions implied by `minimum` restriction level plus the following options:
	`noImageRWNetFile`
	`noOpenAllPanels`
	`noImageLoad`
	`noPanelEdit`
	`noPanelOptions`
	`noPGroupAssignment`

maximum	Combines the restrictions implied by the `intermediate` restriction level plus the following options:
	`noPanelRWConfig`
	`noImageSaving`
	`noPanelAccess`
	`noCMapOpenMap`
	`noCMapSaveMap`

DX*saveImageFormat

Specifies the default file format to be used in the **Print Image** dialog box available in the Image window. The following values are recognized:

PSCOLOR	Color PostScript
PSGREY	Gray level PostScript
EPSCOLOR	Encapsulated Color PostScript
EPSGREY	Encapsulated Gray level PostScript
RGB	“rgb” format
R+G+B	“r+g+b” format
TIFF	TIFF format

DX*saveImagePageSize

Specifies the default page size that PostScript images should be centered on when saving images from the **Save Image** dialog box available from the Image window. Units are those indicated by the metric option.

DX*saveImageResolution

Specifies the default resolution that PostScript images should be saved with when saving images from the **Save Image** dialog box available from the Image window. Resolution is in dots per inch unless the DX*metric resource is set, in which case dots per centimeter is used. This resource is overridden by the saveImageSize resource. If this is set to zero (the default), then the **Save Image** dialog box chooses the resolution, which may be specified inside the visual program.

DX*saveImageSize

Specifies the default size of PostScript images saved from the **Print Image** dialog box available from the Image window. If specified this option overrides the saveImageResolution option. The value is a string in the same format as that accepted by the dialog box (for example, “8”, “8x”, “x10”, “8x10”). Units are those indicated by the metric option. The default value is determined by saveImageResolution option.

DX*vpeCanvas.lineThickness

Specifies the thickness of the lines that connect module outputs to module inputs in the VPE.

DX*warningEnabled

Specifies the default value of the **Warning Messages** toggle button in the Message window’s **Option** menu.

DX*warningOpensMessage

Specifies whether or not the Message window pops up when a warning message is printed in the Message window.

Appendix E. Data Explorer Fonts

All fonts used by Data Explorer are stored as files in the `/usr/lpp/dx/fonts` directory. Users can add their own fonts by creating a file in the correct format and using the DXFONTS environment variable to list the directory or directories where the additional font files are stored.

Fonts are stored in the standard Data Explorer file format. A font file must contain the following information (for more information on the Data Explorer file format, see Appendix B, “Importing Data: File Formats” on page 241).

The font is a group that contains 256 fields, one for each ASCII character value, and two attributes describing the height of the font. For example:

```
object “myfont” class group
   member 0 value “empty”
   member 1 value “control-a”
  .
  :
   member 65 value “A”
   member 66 value “B”
  .
  :
   member 255 value “empty”
   attribute “font ascent” number 0.75
   attribute “font descent” number 0.25
```

The attributes describe the maximum height above and below the baseline for all characters in this font. The values should be positive floating point numbers and they should add to 1.0.

Each member of the group must be a field.

Each field must contain a 2-D or 3-D “positions” component, and a “connections” component with element type “lines” or “triangles”.

```
object “positions1” class array type float rank 1 shape 3  items 3
  data follows  0.1 0.5 0.0   0.3 0.6 0.0    0.5 0.5 0.0

object “connections1” class array type int rank 1 shape 2 items 2
  data follows  0 1   1 2
  attribute “element type” string “lines”
  attribute “ref” string “positions”
```

The character positions are assumed to have their horizontal (X) origin at the left edge and their vertical (Y) origin at the baseline.

Each field must have a “char width” attribute describing the character width for spacing when combining characters into strings.

```
object “circumflex” class field
  component “positions” value “positions1”
  component “connections1” value “connections1”
  attribute “char width” number 0.55
```

The fields can contain other components (such as “colors” or “normals”), although the standard Data Explorer modules will not process this information.

Those ASCII codes for which there is no representation should be empty fields:

```
object “empty” class field
```

A space character can be generated by making an empty field with a positive width:

```
object “wide” class field
  attribute “char width” number 0.3

object “myfont” class group
 .
 :
  member 32 value “wide”
 .
 :
```

Overstrike characters can be generated by making a field with a zero or negative width:

```
object “umlaut” class field
  component “connections” value “con1”
  component “positions”   value “pos1”
  component “char width”  number 0.0
```

The following fonts are supplied with Data Explorer, and may be found in this directory `/usr/lpp/dx/fonts`. Each one is a Data Explorer format file. If you would like to look at the structure of a font file, simply Import the data using the Import module, and Export it in text format.

default fonts	area	an area font (same as pitman)
	fixed	a fixed width font (same as roman_sfix)
	variable	a variable width font (same as roman_s)
cyrilic font	cyril_d	a cyrilic double-line font
Gothic fonts	gothiceng_t	an English gothic triple-line font
	gothicger_t	a German gothic triple-line font
	gothicit_t	an Italian gothic triple-line font
Greek fonts	greek_s	a Greek single-line font
	greek_d	a Greek double-line font
italic fonts	italic_d	an italic double-line font
	italic_t	an italic triple-line font
area (filled) font	pitman	an area typewriter style font that includes European National Language characters
Roman fonts	roman_s	a Roman single-line sans serif font
	roman_d	a Roman double-line sans serif font
	roman_dser	a Roman double-line serif font
	roman_tser	a Roman triple-line serif font

	roman_ext	an extended character set Roman single-line sans-serif font that includes European National Language characters
script fonts	script_s	a script single-line font
	script_d	a script double-line font

The default font directory is `/usr/lpp/dx/fonts`. For user-added fonts, the DXFONTS variable can contain a colon-separated list of directories to search for fonts before searching the default directory. A font file name must be the same as the font name. For example, the font "cursive" should be stored in the "cursive.dx" file, and the **font** parameter to the Caption module should be "cursive" to use this font. The names of user-supplied fonts should be all lowercase.

Since font files are in the standard Data Explorer file format, in addition to being used by Caption, Plot, AutoAxes, and ColorBar, they can be read into Data Explorer with Import and processed like any other group. Individual characters can be selected with Select, colored with Color, and displayed with ShowConnections. Newly constructed fonts can be exported with Export.

Table 8 and Table 9 on page 310 illustrate those characters that are part of the roman_ext font. The appearance of a character may differ from that illustrated in the tables.

Table 8. roman_ext Font Characters (Part 1)

Octal Value	**40**	**60**	**100**	**120**	**140**	**160**	**200**
0	Blank	0	@	P	‘	p	Ç
1	!	1	A	Q	a	q	ü
2	"	2	B	R	b	r	é
3	#	3	C	S	c	s	â
4	$	4	D	T	d	t	ä
5	%	5	E	U	e	u	à
6	&	6	F	V	f	v	å
7	'	7	G	W	g	w	ç
10	(	8	H	X	h	x	ê
11	)	9	I	Y	i	y	ë
12	*	:	J	Z	j	z	è
13	+	;	K	[	k	{	ï
14	,	<	L	\	l	¦	î
15	-	=	M	]	m	}	ì
16	.	>	N	↑	n	˜	Ä
17	/	?	O	_	o	□	Å

Table 9. roman_ext Font Characters (Part 2)

Octal Value	220	240	260	300	320	340	360
0	É	á		Α	Ρ	ι	
1	æ	í		Β	Σ	κ	
2	Æ	ó		Γ	Τ	λ	
3	ô	ú		Δ	ϒ	μ	
4	ö	ñ		Ε	Φ	ν	
5	ò	Ñ		Ζ	Χ	ξ	
6	û	ª		Η	Ψ	ο	
7	ù	º		Θ	Ω	π	
8	ÿ	¿		Ι	α	ρ	
9	Ö			Κ	β	σ	
10	Ü			Λ	γ	τ	
11	¢			Μ	δ	υ	
12	£			Ν	ε	φ	
13	¥	¡		Ξ	ζ	χ	
14	₧	«		Ο	η	ψ	
15	ƒ	»	∫	Π	θ	ω	

Table 10 (Page 1 of 2). Additional Symbols \001 - \035

1	apostrophe curly open	‘
2	apostrophe curly close	’
3	backslash	\
4	bullet	•
5	cent	¢
6	copyright	©
7	cross product	×
10	degree	°
11	emdash	—
12	endash	–
13	exclamation	¡
14	franc	ƒ
15	guillemet open	«
16	guillemet close	»
17	guillemet single open	
20	guillemet single close	
21	infinity	∞
22	integral	∫
23	interrogatory	¿
24	minus	−

Table 10 (Page 2 of 2). Additional Symbols \001 - \035

25	notequal	≠
26	plusminus	±
27	pound	£
30	quote Open	“
31	quote close	”
32	registered	®
33	similar	∼
34	trademark	™
35	yen	¥

The octal values for the English and Greek character sets in the pitman (or area) font are the same as those illustrated in Table 8 on page 309 and Table 9 on page 310. European National Languages characters are provided using single octal values or using a combination of values (see Table 11). For example, ä is produced using `252a`, and î is produced using `\221\237`. Additional special symbols are provided with octal values 001 to 035 (see Table 10 on page 310).

The following illustrates how you could produce a caption that contains the string “Jag är här på semester”.

```
caption = Caption("Jag \252ar h\252ar p\213a semester"; font="pitman");
camera = AutoCamera (caption);
Display(caption,camera);
```

Table 11 (Page 1 of 2). European National Language Symbols and Characters \200 to \255

200	aBar		
201	accentAcute	´	To type a acute, enter \201a. To type i acute, enter\201\237.
202	AccentAcute	´	(Uppercase except I)
203	IAccentAcute	Í	(Uppercase I Acute) To type A acute, enter \202A.
204	accentGrave	`	(lowercase)
205	AccentGrave	`	(Uppercase except I)
206	IAccentGrave	Ì	(Uppercase I)
207	accentHungarian		(Double quote accent over o and u)
210	AccentHungarian		(Double quote accent over O and U)
211	ae	æ	(Ligature)
212	AE	Æ	(Ligature)
213	angstrom		(Lowercase)
214	Angstrom		(Uppercase)
215	breve		(Lowercase)
216	Breve		(Uppercase)
217	c cedilla	ç	(Lowercase c cedilla)

Table 11 (Page 2 of 2). European National Language Symbols and Characters \200 to \255

220	C cedilla	Ç	(Uppercase C cedilla)
221	widecircumflex	^	(Looks better than ASCII Circumflex when used as a lowercase accent)
222	WideCircumflex	^	(Uppercase except for I and O)
223	WideCircumflex	Î	(Uppercase I)
224	WideCircumflex	Ô	(Uppercase O)
225	Clicka		(Used in Lithuanian over C, S, Z, c, s, and z; also called *hacek* or *caron*)
226	CommaRomanian		(Used in Romanian under S, T, s, and t)
227	enya		(Lowercase for Spanish n, and Portuguese a and o)
230	Enya		(Uppercase for Spanish N, and Portuguese A and O)
231	eth	ð	(Icelandic)
232	Eth	Đ	(Icelandic)
233	hookLithuanian		(Cedilla-like hook used under Lithuanian a, e, and u)
234	ihookLithuanian		(Cedilla-like hook used under Lithuanian i)
235	HookLithuanian		(Cedilla-like hook used under Lithuanian A, E, and U)
236	IHookLithuanian		(Cedilla-like hook used under Lithuanian I)
237	i dotless		(Dotless i to be accented with acute, grave, etc.)
240	l slash		(Slashed l used in Polish)
241	L slash		(Slashed L used in Polish)
242	macron		(Lowercase)
243	Macron		(Uppercase)
244	oBar		
245	o slash	ø	(Danish slashed o)
246	O slash	Ø	(Danish slashed O)
247	Overdot	·	(Dot placed over Polish Z and z)
250	thorn	þ	(Icelandic)
251	Thorn	Þ	(Icelandic)
252	umlaut	¨	(Lowercase, also called diaresis)
253	Umlaut	¨	(Uppercase, except for I and O)
254	IUmlaut	¨	(I umlaut)
255	OUmlaut	¨	(O umlaut)

Appendix F. Data Explorer Colors

The following list of defined colors is internal to Data Explorer and is accessed when the `colors.txt` file is unavailable (see "Color" on page 75 in *IBM Visualization Data Explorer User's Reference*).

Data Explorer accepts these names as valid strings for specifying colors to a visual program. Each color is followed by the corresponding RGB vector.

aquamarine	0.4392157	0.8588235	0.5764706
black	0.0000000	0.0000000	0.0000000
blue	0.0000000	0.0000000	1.0000000
blueviolet	0.6235294	0.3725490	0.6235294
brown	0.6470588	0.1647059	0.1647059
cadetblue	0.3725490	0.6235294	0.6235294
coral	1.0000000	0.4980392	0.0000000
cornflowerblue	0.2588235	0.2588235	0.4352941
cyan	0.0000000	1.0000000	1.0000000
darkgreen	0.1843137	0.3098039	0.1843137
darkolivegreen	0.3098039	0.3098039	0.1843137
darkorchid	0.6000000	0.1960784	0.8000000
darkslateblue	0.4196078	0.1372549	0.5568628
darkslategray	0.1843137	0.3098039	0.3098039
darkslategrey	0.1843137	0.3098039	0.3098039
darkturquoise	0.4392157	0.5764706	0.8588235
dimgray	0.3294118	0.3294118	0.3294118
dimgrey	0.3294118	0.3294118	0.3294118
firebrick	0.5568628	0.1372549	0.1372549
forestgreen	0.1372549	0.5568628	0.1372549
gold	0.8000000	0.4980392	0.1960784
goldenrod	0.8588235	0.8588235	0.4392157
gray	0.7529412	0.7529412	0.7529412
green	0.0000000	1.0000000	0.0000000
greenyellow	0.5764706	0.8588235	0.4392157
grey	0.7529412	0.7529412	0.7529412
indianred	0.3098039	0.1843137	0.1843137
khaki	0.6235294	0.6235294	0.3725490
lightblue	0.7490196	0.8470588	0.8470588
lightgray	0.8274510	0.8274510	0.8274510
lightgrey	0.8274510	0.8274510	0.8274510
lightsteelblue	0.5607843	0.5607843	0.7372549
limegreen	0.1960784	0.8000000	0.1960784

magenta	1.0000000	0.0000000	1.0000000
maroon	0.5568628	0.1372549	0.4196078
mediumaquamarine	0.1960784	0.8000000	0.6000000
mediumblue	0.1960784	0.1960784	0.8000000
mediumforestgreen	0.4196078	0.5568628	0.1372549
mediumgoldenrod	0.9176471	0.9176471	0.6784314
mediumorchid	0.5764706	0.4392157	0.8588235
mediumseagreen	0.2588235	0.4352941	0.2588235
mediumslateblue	0.4980392	0.0000000	1.0000000
mediumspringgreen	0.4980392	1.0000000	0.0000000
mediumturquoise	0.4392157	0.8588235	0.8588235
mediumvioletred	0.8588235	0.4392157	0.5764706
midnightblue	0.1843137	0.1843137	0.3098039
navy	0.1372549	0.1372549	0.5568628
navyblue	0.1372549	0.1372549	0.5568628
orange	0.8000000	0.1960784	0.1960784
orangered	1.0000000	0.0000000	0.4980392
orchid	0.8588235	0.4392157	0.8588235
palegreen	0.5607843	0.7372549	0.5607843
pink	0.7372549	0.5607843	0.5607843
plum	0.9176471	0.6784314	0.9176471
red	1.0000000	0.0000000	0.0000000
salmon	0.4352941	0.2588235	0.2588235
seagreen	0.1372549	0.5568628	0.4196078
sienna	0.5568628	0.4196078	0.1372549
skyblue	0.1960784	0.6000000	0.8000000
slateblue	0.0000000	0.4980392	1.0000000
springgreen	0.0000000	1.0000000	0.4980392
steelblue	0.1372549	0.4196078	0.5568628
tan	0.8588235	0.5764706	0.4392157
thistle	0.8470588	0.7490196	0.8470588
turquoise	0.6784314	0.9176471	0.9176471
violet	0.3098039	0.1843137	0.3098039
violetred	0.8000000	0.1960784	0.6000000
wheat	0.8470588	0.8470588	0.7490196
white	1.0000000	1.0000000	1.0000000
yellow	1.0000000	1.0000000	0.0000000
yellowgreen	0.6000000	0.8000000	0.1960784

Appendix G. Accelerator Keys

The following table is a summary of the accelerator keys available in Data Explorer. To learn how to use these keys and their functions, see “Selecting Pull-Down Menus and Pull-Down Menu Options” on page 62.

Table 12. Summary of Data Explorer Accelerator Keys

Function	Accelerator Key	Active Primary Window				
		VPE Window	**Image Window**	**Control Panel**	**Colormap Editor**	**Message Window**
Add Input Tab	Ctrl+A	√				
Camera mode	Ctrl+K		√			
Configuration	Ctrl+F	√				
Cursors mode	Ctrl+X		√			
Delete	Ctrl+Delete	√		√		
Delete Selected Control Points	Ctrl+Delete				√	
End Execution	Ctrl+End	√	√	√		√
Execute on Change	Ctrl+C	√	√	√		√
Execute Once	Ctrl+O	√	√	√		√
Hide All Tabs	Ctrl+H	√				
Navigate mode	Ctrl+N		√			
Next Error	Ctrl+N					√
Open All Control Panels	Ctrl+P	√				
Open Colormap Editor	Ctrl+E	√				
Open All Colormap Editors	Ctrl+E		√			
Pan/Zoom mode	Ctrl+G		√			
Pick mode	Ctrl+I		√			
Previous Error	Ctrl+P					√
Quit/Close	Ctrl+Q	√	√		√	√
Redo	Ctrl+D		√			
Remove Input Tab	Ctrl+R	√				
Reset	Ctrl+F		√			
Reveal All Tabs	Ctrl+L	√				
Roam	Ctrl+W		√			
Rotate mode	Ctrl+R		√			
Save	Ctrl+S	√	√			
Select All Control Points	Ctrl+A				√	
Tool Palettes	Ctrl+T	√				
Undo	Ctrl+U		√			
View Control	Ctrl+V		√			
Zoom mode	Ctrl+Z		√			

Glossary

Some of the definitions in this glossary are taken from the *IBM Dictionary of Computing*, SC20-1699.

A

accelerator. A “shortcut” that minimizes the number of keystrokes or mouse clicks required to complete a task.

anchor window. The window in which a Data Explorer session starts (either the Visual Program Editor or the Image window). The window is identified by an anchor symbol in the top left corner. When this window is closed, the Data Explorer session ends.

architecture. The organizational structure of a computer system, including hardware and software.

array. In Data Explorer, an array structure containing an ordered list of data items of the same type along with additional descriptive information. Arrays are either compact or irregular. See *compact array, irregular array*.

assembly. An object representing a collection of objects.

attribute. A characteristic of an object. Objects can have attributes that are indexed by a string name and have a value that is an object. See also *component attribute*.

C

camera. An object that describes the viewing parameters of an image (e.g., width of the viewport, viewer’s location relative to the object, and the resolution and aspect ratio of the image). A camera may be explicitly defined and passed as a parameter to the Render or Display module. It may also be implicitly defined in the use of interactive, mouse-driven options (such as zoom or rotate) in the Image window.

canvas. The area of a VPE window used in building and editing visual programs.

cell-centered data. Connection-dependent data.

clipping plane. A plane that divides a three-dimensional object into a rendered and an unrendered region, making the object’s interior visible.

colormap. A map that relates colors to data values. The colors are carried in the map’s “data” component and the data values to which each color applies in its “positions” component.

colormap editor. A special tool for mapping precise colors to specified data values, the results of which are displayed in a visual image.

compact array. Any of five types of compact encoding of array data:

constant array
mesh array
path array
product array
regular array

component. A basic part of a field (such as “positions,” “data,” or “colors”); each component is indexed by a string (e.g., “positions”), and its value is typically an array object (e.g., the list of position values). See also *component attribute*.

component attribute. A characteristic of a component. Components of a field can have attributes that are indexed by a string name and have a value that is an object.

composite field. A grouping of like fields for processing a single spatial entity. See also *partitioned field*.

connection. Component of an IBM Data Explorer data field that specifies how a set of points are joined together. Also controls interpolation.

connection-dependent data. Cell-centered data. The data value is interpreted as constant throughout the connection element.

contour. On a surface, a line that connects points having the same data value (e.g., pressure, depth, temperature).

control panel. An IBM Data Explorer window that facilitates setting and changing the parameters of visual programs.

cube. A volumetric connection element that connects eight positions in a data field.

cutting plane. An arbitrary plane, in three-dimensional space, onto which data are mapped.

D

data-driven interactors. Interactors whose attributes (such as minimum and maximum) are set by an input data field.

Data Prompter. An interface that enables a user to describe the format of the data in a file. The prompter creates a General Array Format header file that is used by the Import module to import the data.

dependence. A component attribute. One component is said to be dependent (“dep”) on another if the items in their component arrays are in one-to-one correspondence to each other.

dialog box. The “window” displayed when the user selects a pull-down option that offers or requires more detailed specification.

display. (1) To present information for viewing, usually on a terminal screen or a hard-copy device. (2) A device or medium on which information is presented, such as a terminal screen. (3) Deprecated term for *panel*.

E

element. Connection item.

element type. An attribute that describes the type of connection element, for example, “cubes”, “tetrahedra”, or “lines”.

executive. The component of the Data Explorer system that manages the execution of specified modules. The term often refers to the entire server portion of the Data Explorer client-server model, including the executive, modules, and data-management components.

F

face. (1) Any planar surface that bounds a three-dimensional object. (2) A polygon.

field. A self-contained collection of data items. A Data Explorer field typically consists of the data itself (the “data” component), a set of sample points (the “positions” component), a set of interpolation elements (the “connections” component), and other information as needed.

flat shading. A shading model in which each face of an object is shaded with a single intensity value. Contrast with *Gouraud shading*.

fork. An operation that causes a program to branch into two or more parallel concurrent paths.

fork-join parallelism. A programming mechanism that supports parallel processing: The fork statement splits a single computation into multiple independent computations. The join statement recombines two or more concurrent computations into one.

G

general array format. A data-importing method that uses a header file to describe the data format of a data file. This “format” makes it possible to import data in a variety of formats.

glyph. A graphical figure used to represent values of a particular variable. The length, angle, or other attribute of the glyph is some function of the value of that variable. Each occurrence of a glyph represents a single value of the variable.

Gouraud shading. Also called intensity interpolation shading. A shading model in which the intensity of values of incident illumination on a polygon are interpolated from intensity values at the vertices of the polygon. Contrast with *flat shading*.

graphical user interface. A set of panels and dialogs for interacting with an application.

group. A collection of objects.

I

icon. A displayed symbol that a user can point to with a device such as a mouse to select a particular operation or software application.

image window. IBM Data Explorer window that displays the image generated by a visual program. Associated with the Image window are special interactors for 3-D viewing.

interactor. A Data Explorer device used to manipulate data in order to change the visual image produced by a program. See also *data-driven interactor, interactor stand-in*.

interactor stand-in. An icon used in the VPE window to represent an interactor. Stand-ins are named after the type of data they generate:

- integer
- scalar
- selector (outputs a value and a string)
- string
- value
- vector

interpolation element. An item in the connections component array. Each interpolation element provides a means for interpolating data values at locations other than the specified set of sample points. See *positions component*.

invalid. A classification of an array item (typically positions or connections). An invalid item is not to be rendered or realized.

irregular array. In contrast to a compact array, an array in which the data is stored explicitly.

isosurface. A surface in three-dimensional space that connects all the points in a data set that have the same value.

isovalue. The single value that characterizes each and every point constituting an isosurface. By default, this value is the average of all the data values in the set being visualized.

item. A single piece of data in an array.

J

join. An operation that merges two or more computation paths.

L

line. An element that connects two positions in a field.

M

macro. In IBM Data Explorer, a sequence of modules that acts as a functional unit and is displayed as a single icon. Macros can also be defined in the Data Explorer scripting language.

member. An individual unit or object in a group. A collection of members makes a group.

menu bar. In windows, a horizontal bar that displays the names of one or more menus (or tasks). When the user selects a menu, a pull-down list of options for that menu is displayed.

mesh array. A compact array that encodes multidimensional regularity of connections. It is a product of path arrays. In a mesh array, which positions are connected to one another is implicitly rather than explicitly defined.

module. (1) In IBM Data Explorer, a primitive function, such as Isosurface. (2) In a VPE window, the icon for a module. (3) A program unit that is functionally discrete and identifiable (i.e., it can be assembled, compiled, combined with other units, and so on).

N

navigate. To move the camera (changing the “to” and “from” points) around the image scene, using the mouse.

netCDF. Network Common Data Form.

network. In Data Explorer the set of tool modules, interactor stand-ins, and connections that constitute a visual program. In the VPE window, a network appears as a set of icons connected by arcs.

Network Common Data Form (netCDF). A data format that stores and retrieves scientific data in self-describing, multidimensional blocks (netCDF is not a database management system, however). netCDF is accessible with C and FORTRAN.

normal. (1) Perpendicular to a surface. (2) In IBM Data Explorer, a vector that is perpendicular to a face or surface of an object. A normal may depend on connections or positions. A connection-dependent normal results in flat shading; a position-dependent normal results in Gouraud shading.

O

object. In IBM Data Explorer, any discrete and identifiable entity; specifically, a region of global memory that contains its own type-identification and other type-specific information.

opacity. The capacity of matter to prevent the transmission of light. For a surface, an opacity of 1 means that it is completely opaque; an opacity of 0, that it is completely transparent. For volume, opacity is defined as the amount of attenuation (of light) per unit distance.

P

page. (1) That portion of a panel displayed by a user interface. (2) To move back and forth through the pages of a multipage panel. See also *scroll*.

palette. A displayed grouping of available selections (such as functions, modules, or colors) in a user-interface window.

palindrome. In Data Explorer, a mode of running an entire sequence, first in one direction, then in the opposite direction.

panel. A formatted display of information on a display screen. See also *window*. Synonymous with *display*.

partitioned field. A composite field, created by partitioning a single field into a collection of separate fields; used for parallel processing and data-management purposes.

path array. A compact array that encodes linear regularity of connections. It is a set of *n*–1 line segments, where the *i*th line segment joins points *i* and *i* + 1.

pixel. Picture element. In computer graphics, the smallest element of a display surface that can be independently assigned color and intensity.

polygon. (1) Any multisided planar figure. (2) A face of a three-dimensional object.

position-dependent data. Data that are in one-to-one correspondence with positions.

positions component. A component that consists of a set of dimensional points in a field.

probe. A list of one or more vectors that represent points in a graphical image. Probes can be used with Data Explorer tools that accept vectors as input (such as ClipPlane and Streamline) or to control the view of an image.

product array. A compact array that encodes multidimensional positional regularity. It is the set of points obtained by summing one point from each of the terms in all possible combinations. In the simplest case, each term is a regular array.

pull-down. In windows, the list of options displayed when a task is selected from the menu bar.

Q

quad. An element that connects four positions in a field.

R

rank. The number of dimensions in an array. Rank zero corresponds to scalars (e.g., the number 3). Rank one corresponds to vectors (e.g, *[1.5 3.7]* and *[2.9 4.0 6.0]*). Rank two corresponds to matrices or rank-two tensors (e.g., the matrix *[[1 3 8][5 7 2][1 0 1]]*). Higher ranks correspond to higher-order tensors. See also *shape*.

realization. A description of how raw data is to be represented in terms of boundaries, surfaces, transparency, color, and other graphical, image, and geometric characteristics.

reference. A component attribute. One component is said to *refer to* another ("ref") if the items in the first array are integer indices into the second array. The connections component references the positions component.

regular array. A compact array that is a set of *n* points lying on a line, with constant spacing between them, which can represent one-dimensional regular positions.

rendering. The generation of an image from some representation of an object, such as a surface, or from volumetric information.

ribbon. A figure derived from lines (e.g., from streamlines and streaklines). Ribbons may twist to indicate vorticity.

S

sample point. A point that represents user data. Data is interpolated between sample points by interpolation elements (connections).

scalar. A non-vector value characterized by a single, real number.

scatter data. A collection of sample points without connections.

screen. An illuminated display surface (e.g., the display surface of a CRT or plasma panel).

scripting language. The IBM Data Explorer command language. Used for writing visual programs, to manage the execution of modules, and to invoke visualization functions.

scroll. To move all or part of the display image vertically or horizontally to display data that cannot be observed in a single display image.

secondary window. Any window generated by another window. A secondary window always appears on top of its parent window and is automatically minimized or closed when the parent window is minimized or closed. Synonymous with *child window*.

sequencer. An IBM Data Explorer tool for creating "animated" sequences of images.

series. In IBM Data Explorer, used to represent a single field sampled across some parameter (e.g., a simulation of a CMOS device across a temperature range). Members of a series have a position. A copy of the position is found in the "series position" attribute.

shape. A list of the dimensions of a structure (the list contains nothing for scalars, one entry for vectors, two for rank-two tensors, and so on). See also *rank*.

shared. A term used to indicate the availability of a resource for use by more than one program at the same time.

specular reflection. A reflection from a shiny object.

stand-in. See *interactor stand-in*.

streaklines. Lines that represent the path of particles in a changing vector field. Also called rakes.

streamlines. Lines that represent the path of particles in a vector field at a particular time. Also called flow lines.

T

task. A basic unit of work to be accomplished by a computer.

tetrahedron. A volumetric connection element that connects four positions in a field.

tool. In IBM Data Explorer, a general term for any icon used to build a visual program (specifically, module, macro, or interactor stand-in).

triangle. A connection element that connects three points in a field.

tube. A surface centered on a deriving line (e.g., a streamline or streakline). Tubes may twist to indicate vorticity.

U

user. Anyone who uses the services of a computer system. See also *user display station*.

V

value. An instance of an attribute (for example, “blue” as the value of the attribute “color”).

vector. A quantity characterized by more than one component.

visual program. A user-specified interconnected set of Data Explorer modules that performs a sequence of operations on data and typically produces an image as output.

vertex. One of the positions that define a connection element.

visual program editor. IBM Data Explorer window used to create and edit visual programs and macros. See also *canvas*.

volume. The amount of three-dimensional space occupied by an object or substance (measured in cubic units). To be distinguished from an object’s surface, which is a mathematical abstraction.

volume rendering. A technique for using color and opacity to visualize all the data in a 3-dimensional data set. The internal details visualized may be physical (such as the structure of a machine part) or they may be other characteristics (such as fluid flow, temperature, or stress).

vorticity. Mathematically defined as the curl of a velocity field. A particle in a velocity field with nonzero vorticity will rotate.

W

window. On a visual display terminal, information in a framed area on a panel that overlies part of the panel. See also *anchor window, primary window, secondary window*.

wireframe. Connected lines that represent a surface.

Index

Readers' Comments — We'd Like to Hear from You

IBM Visualization Data Explorer
User's Guide
Version 3 Release 1 Modification 4

Publication No. SC38-0496-06

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	□	□	□	□	□

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? □ Yes □ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You
SC38-0496-06

Cut or Fold Along Line

IBM®

Fold and Tape

Please do not staple

Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Thomas J. Watson Research Center/Hawthorne
Data Explorer Development
P.O. Box 704
YORKTOWN HEIGHTS, NY
USA 10598-0704

Fold and Tape

Please do not staple

Fold and Tape

SC38-0496-06

Cut or Fold Along Line

Printed in U.S.A.

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber

SC38-0496-06