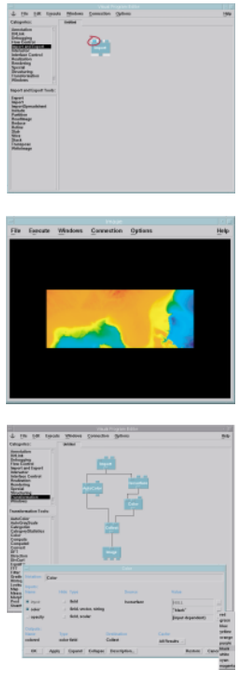
Paths to Visualization is the first book to provide training materials for OpenDX, the open source version of the former IBM product *Visualization Data Explorer*™. The material presented here has been tested and refined in years of actual training sessions, and is currently used by VIS, Inc. for their OpenDX beginners course. The material is organized and presented as a sequence of lessons, designed to facilitate self-paced instruction for students working independently or in the context of a formal class. Each lesson is built around a visualization goal, an appropriate data model, clearly identified opportunities for individual experimentation and refinement, an example step-by-step solution, and discussion of relevant concepts. The visualization concepts covered apply in general, but are specifically illustrated with the facilities and operation of OpenDX. The set of lessons is designed to systematically take the serious student from the level of OpenDX novice to that of the advanced beginner, with each new lesson building upon concepts, examples, or visual programs developed in preceding lessons. The materials assume that the student is working *hands-on* with the OpenDX Visual Programming Environment, with access to real example data sets. It also provides hand-on experience with a variety of data sets, helping the student to understand how to import differently structured data and how to exploit the underlying OpenDX data model to achieve desired visual effects.

WHAT YOU NEED TO USE THIS BOOK:

- OpenDX software version 4.1.0 or later -- the software can be downloaded free from <http://www.opendx.org/>
- Sample data files available from VIS, Inc., which can be downloaded free from <http://www.vizsolutions.com/>



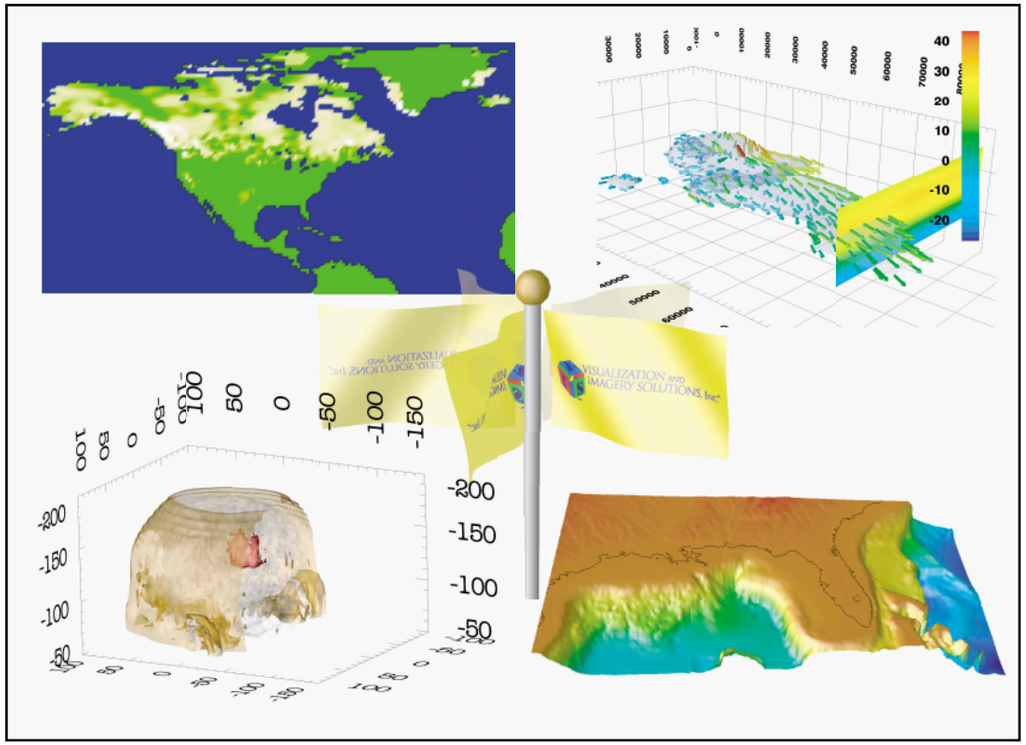
The lessons, with step-by-step solutions, help you understand how OpenDX can be used to visualize your data. They lead you through uses of OpenDX tools and example data sets, illustrating the various visual possibilities supported by OpenDX.

OpenDX Paths to Visualization

VIS INC.

OpenDX

Paths to Visualization



David Thompson • Jeff Braun • Ray Ford

OpenDX

Paths to Visualization

Second Edition

OpenDX

Paths to Visualization

Second Edition

Materials used for learning OpenDX–the open source derivative of IBM's Visualization Data Explorer.

VISUALIZATION AND IMAGERY SOLUTIONS, INC.™

Visualization and Imagery Solutions, Inc.
5515 Skyway Drive
Missoula, MT 59804

David L. Thompson Jeff A. Braun Ray Ford

Publisher: Visualization and Imagery Solutions, Inc.
Product Manager: David Thompson
Copyeditor: Ray Ford
Cover Designer: David Thompson

Background

VIS Inc. is a company founded by three veteran "power users" of IBM's **Visualization Data Explorer** (DX) to provide third party support to DX users as the software makes the transition from a commercially developed and supported product into "open-source". The VIS Inc. principals have combined expertise of over 40 years in software development, and over 15 years of DX expertise. They have never been IBM employees nor official DX developers, but they have been aggressive DX users since 1992. Their primary interest in DX has always been to produce high-quality visual products. Given this goal, over the years they have developed secondary interests in building tools that enhance their own productivity and creating training materials that assist other users and developers in becoming productive with DX. Their primary interest now is to keep the DX system alive and viable as a visualization system option through active development and support, in a manner compatible with the open-source software model.

As long time users of DX, the VIS Inc. principals have formed close working relationships with the original DX developers and project managers at IBM, as well as with users all over the world. The principals have also made significant contributions to both the training of DX users worldwide and the body of DX-related software that is openly available. With the withdrawal of IBM's active support for DX as a commercial product, VIS Inc. was formed to help provide third party expertise needed to assure that DX remains a viable open-source alternative for commercial and government users. That is, VIS Inc. hopes to provide value-added service and support for the DX users who wish such assistance, in a manner analogous to that popularized by third party value-added service providers for Linux such as RedHat Inc., Caldera Inc. and O'Reilly and Associates.

The most critical element of third party, value-added support for an open source system is expertise with that system. VIS Inc. brings together a group of some of the most experienced DX users and developers outside of the core DX development group. In their combination of over 15 years experience with DX, the VIS Inc. principals have made the following contributions.

* Applications Development. They have used DX as the primary visualization tool in a range of multi-disciplinary research projects, particularly in applications involving natural resource applications. For a list of related publications and visual artifacts, see http://www.cs.umt.edu/Dxcontrib.

* Instruction and Training. They have used DX since 1992 as the primary software in interdisciplinary, graduate level university courses on data visualization. They have also offered non-credit, specialized training courses at their own local training site and at customers' sites around the country. The VIS Inc. principals include two of the four internationally certified DX trainers used by the IBM Visualization Data Explorer Group to staff IBM's training sessions.

* DX-ware. They have developed a number of special purpose and general interest DX enhancements, useful in a wide variety of visual applications. These have been freely distributed from several public DX web sites. These include colormaps, macros, modules and data importers.

* "GIS-2-DX". Under contract from IBM, they developed and have supported a widely used software package that allows data sets in standard, external formats to be easily imported into DX, i.e., the "gis2dx" package freely distributed by IBM since 1993.

* Visualization Consulting. They have worked as consultants to a variety of industry partners on the use of DX in a broad range of example applications, including remotely sensed data analysis, geological/geophysical data depiction, interface with data from geographical information systems (GIS), hydrological and ground water flow, landscape and ecological modeling and analysis, and wildfire modeling.

Even a few years ago, IBM's decision to withdraw its support would have meant the death of DX as a software system, forcing users to seek other (we think, less satisfactory) alternatives. But that situation has changed with the demonstration, through examples such as Linux, Apache, and others, that open-source software systems can and will remain viable if the software base is sound, if there is a critical mass of committed users, and if there are third party entities willing to commit to a high quality of service and support. We think that **OpenDX** and its current user base clearly satisfy the first two criteria, and we have committed VIS Inc. to satisfying the third by providing a wide range of support, training, development, and advanced consulting on the use of **OpenDX**.

Contact us to let us know how we can help satisfy your concerns and/or future needs for **OpenDX**.

With **special thanks** to Keith Sams, Eric Nakata, and the rest
of the IBM Visualization Data Explorer Team

Table of Contents

1 How to Use This Material

Information about the Material

This material is written to be used as a "hard-copy companion" for new users of OpenDX, learning OpenDX in one of several modes: as part of an organized training class, supervised and supported by OpenDX training staff; working semi-independently but with remote (via phone or email) contact with training and support staff; or working truly independently with the only support being access to on-line documentation, examples, mailing lists, etc. To support all these modes, the material is designed to be self-contained, based on a sequence of examples that lead the student from basic to more complex concepts, through the incremental development of first basic, then more complex visualizations.

The material is logically organized as a sequence of lessons, with each new lesson building upon concepts, examples, and/or visual programs developed in preceding lessons. The lessons are designed to facilitate self-paced instruction for students, whether working independently or in the context of a class, by providing within each lesson a basic visual model and clearly identified opportunity for individual experimentation and refinement.

Styles used in this material

The instructional material uses specialized formatting to direct the user's actions. The following applies:

- DX Modules are shown in a bold typeface such as: **Isosurface** module.
- DX Menu items are shown with underlines such as: File menu.
- DX Input parameters are shown in italics such as: *filename* parameter.
- Clickable buttons in OpenDX are shown with a dotted underline such as: OK button.

Lessons/Chapters

This material is organized into lessons/chapters that are cumulative; thus the training materials should be completed in order. A brief description of the chapters is given below, which includes the approximate time that should be required to complete the exercises in the chapter.

Chapter 2 The Complete Visualization Environment - The process to visualize data with the OpenDX software system is similar to the process used with other visualization systems. The visualization process includes both software independent and software dependent steps that make visualization possible. Approximately 45 minutes to complete.

Chapter 3 First Hands-on Demonstration - A strong beginning can build a foundation that makes software easier to use. This chapter introduces the student to the user interface by having the student create sample visualizations. Working through examples that encapsulate uses of the software helps users build a foundation for understanding and extending the system in other applications. Approximately 2 hours 30 minutes to complete.

Chapter 4 Second Hands-on Demonstration - OpenDX was designed to facilitate working with three-dimensional data sets. As programs (networks) get large, the programmer can use the power of the visual program environment to organize and document programs and their components. Approximately 2 hours to complete.

Chapter 5 First Independent Exercises - The student should now begin to feel comfortable enough to take on a small task without the step-by-step instructions. Approximately 45 minutes to complete.

Chapter 6 Mystery Data - Information embedded inside a data file can be used to build import facilities for these data sets. In this chapter, the student is challenged to import a set of data files for which complete information is not available in advance, and without step-by-step tutorial instructions. After importing the data, the student should follow the program presented at the end of the chapter to learn more about how the data set is organized within OpenDX. Approximately 45 minutes to complete.

Chapter 7 OpenDX Data Model - Understanding how OpenDX organizes and processes data gives users insight how to construct their own visual programs. This chapter gives a brief introduction to the structure of the data model, using a small example to illustrate the interoperability of modules in OpenDX. This material can be used as a reference during later discussions of the data model. Approximately 15 minutes to complete.

Chapter 8 Manipulating Data - Scientific data usually is available in a format determined by the scientific investigation and not by visualization needs. Thus, it is crucial to be able to

manipulate the data inside the visualization environment. Approximately 1 hour to complete.

Chapter 9 More on Data Import - Knowing a data set's structure and being able to describe that structure allows the researcher to import a wide range of data formats. OpenDX allows the visualization user to describe data formats in two different but general manners. The approximate time to read through this chapter is 20 minutes.

Chapter 10 Network Flow Control - OpenDX allows users to construct visualizations in a visual programming environment that combines elements of data flow with conditional flow control. OpenDX's flow control modules allow a program to branch, loop, route, and switch. Approximately 1 hour 30 minutes to complete.

Chapter 11 Series, Categorical, and Scattered Data - OpenDX can accommodate multiple types of data and supports a range of different techniques for working with different data types. This chapter gives three examples to illustrate a few of these techniques. Approximately 1 hour to complete.

Chapter 12 Looping and Probing - The programming interface of OpenDX includes extensions that provide both conditional execution tools and looping tools. It is important to understand both of these control constructs since they provide extended functionality. OpenDX allows the user to interact with and investigate the data set within the Image window using the special Probe and Pick tools. Approximately 45 minutes to complete.

Chapter 13 Tips, Tricks, and Memory Usage - This chapter provides tips for working with large visual programs, rendering images and minimizing memory usage. It also discusses topics such as anti-aliasing, locating modules, Executive memory caching, and some system optimization techniques. The approximate time to read this chapter is 20 minutes.

Chapter 14 Camera Animation and Arranging Images – OpenDX makes it is easy to combine multiple rendered images together into a single conglomerate using prewritten modules such as Arrange. However, it may be more effective to produce an animation to show different view angles. Approximately1 hour to complete.

Chapter 15 Constructing a Native DX File - Using an example of data collected from the Internet, this chapter shows why some data cannot be imported into OpenDX using the standard importing techniques. It then walks the user through the process of understanding the native file format and how it can be used to handle much more sophisticated data organization. Approximately 1 hour to complete.

Chapter 16 Conclusion

The Complete Visualization Environment 2

The Origin and Conceptual Basis of OpenDX

OpenDX originated as a software product known as IBM Visualization System's "Visualization Data Explorer", or Data Explorer, or simply DX. This software was designed, marketed, and supported by IBM Visualization Systems as a product supported on all commercially available Unix workstations. It provides general-purpose, yet specialized, software to support the production of data visualization and analysis. That is, DX is a specialized software system designed only to support visualization, not other types of programming or analysis. However, within the visualization niche DX is general purpose. It supports a broad range of facilities useful in the widest possible range of visualization applications, and is not tailored or customized to the more specific needs of any one limited application domain.

The OpenDX visualization environment is conceptually based on an underlying abstract data model, supported by three powerful visual programming support components. The first programming component is a graphical program editor that allows a user to create a *visual program* using a point and click interface to select program components, designate the order of their application, and define any parameter values they require. Second is a core set of supplied data transformations, each defined and encapsulated as a OpenDX *module* that takes specified inputs, has other user-defined parameters, implements a specific data action, and outputs specific results. The third programming component implements user control over the computation of the visual program, based on a data-flow driven *client-server* execution model. In a simple single-processor execution mode, this facility allows the user to follow program execution by tracing the data flow. In a more computationally-intensive application, this approach allows the visualization to be divided into subcomponents that can be parceled out for execution on multiple workstations or to the multiple processing elements of a modern supercomputer. The client-server execution model allows the user to easily distribute elements of the computation to multiple compute elements. This obviously helps to reduce overall processing time. More importantly, it can dramatically expand the size of data sets that can be effectively processed.

In visual programming terms, the OpenDX environment is designed to allow users to visualize both observed and simulated data, with minimal programming activity. Developers can use the supplied facilities to quickly create *visual programs* that provide imagery, along with interactive controls that allow

users to directly manipulate the image display. For more advanced users, OpenDX also supplements the basic visual programming interface with other, more advanced features: an encapsulation facility that allows program components to be grouped together for use as a *macr*o in other programs; a high level program scripting language; and a full application programming interface which provides support for error handling, user defined data transformations, and an interface to externally written code.

All the visual program development components are based on a very general, application-independent core data model supported by OpenDX. In essence, this data model is an N-dimensional abstract data space from which the OpenDX user takes 2-D and 3-D visual "snapshots" to create viewable images. This is in sharp contrast to more constrained data models that support only a 2-D or 3-D base model, onto which users must fit their data. The OpenDX data model is also purposely defined in a manner independent from particular encodings and data file formats. This distinction between the logical data model and the intricacies of particular file formats allows OpenDX to be flexible and adaptable, supporting *data import* from most applications and formats. Native OpenDX import facilities support various ways to read scientific data sets, allowing the data to be described by their dimensionality, value-type (e.g., real, complex, scalar, vector), location in space, and relationship to other data points. The data model allows details associated with the implementation of data formats and data storage to be hidden from the user. Unlike most of the facilities oriented toward specific graphics capabilities, such as OpenGL program libraries, the OpenDX data model is fully supported as a file based format for import and export. For formats other than the native OpenDX format, native import routines, data description utilities, and data conversion facilities are also included.

A result of OpenDX's data model is that most operations, encapsulated as OpenDX *modules*, appear to be generic transformations that work with a variety of different data types. The operations are interoperable and appear typeless to the user. Furthermore, operations on the data model scale well to the use of multiprocessor architectures to handle very large data sets. On symmetric multiprocessor systems, intramodule parallelism is supported through a simple fork-join shared-memory paradigm. On a collection of networked workstations, parallelism is supported by using the workstations as distributed servers (with a master-slave relationship) to minimize intra-workstation communication and distributed process management. Thus, the OpenDX data model is a key element in OpenDX's ability to scale to the processing of large data sets, to work with a variety of kinds of data, and to work effectively in a range of different processing environments. These benefits all accrue because of the base, general purpose model established in the initial system design. More than anything else, this is what sets OpenDX apart from less general visualization systems, typically designed to support only a particular type of 2-D or 3-D application.

OpenDX Executive and OpenDX Visual Program Editor

The OpenDX Visual Program Editor (User Interface) and Executive are the heart of OpenDX. Typically, an application is built using the point-and-click user interface to construct a visual program. Modules, which may be thought of as subprograms or encapsulated data transformations, are placed on a "programming canvas" and linked together to create the visual program. The OpenDX Executive is a separate process that manages the data flow, determines execution order, and performs the data processing defined by the collection of modules. The Executive is also accessible directly (without the user interface) through the use of advanced programming facilities such as the scripting language. The scripting language supports a more-traditional programming style, in the form of a linear, textual expression of the code. Expressing a program as a script allows additional freedom for the Executive to run programs as batch processes, independent of the visual programming display. However, both the visual programming interface and the scripting language access the same functional modules.

OpenDX Help System and Samples

The Help system and Visualization Samples provide extensive information about OpenDX. There is an on-line tutorial that leads the user step by step through a variety of different example problems. Extensive on-line, context-sensitive help is available. For example, the user can simply click on a module to obtain a "manual page" for the module. The entire set of documentation is also available in html and pdf format for browsing. Finally, there is a large set of sample visual programs and sample data sets, which range from very simple to very complex.

OpenDX Data Prompter

The OpenDX Data Prompter provides a point and click interface which allows the user to easily import a variety of different data formats, including many application-specific, private formats. The Data Prompter also includes a set of general-purpose default visual programs which can be used to depict a data set once it has been imported successfully. This facility combining simple import and display facilities allows a developer to get most data sets imported and have some sort of visualization up running quickly.

OpenDX Development API and DXLink

OpenDX includes a set of libraries corresponding to an application programming interface which allows developers to extend the functionality of OpenDX by adding their own modules. Developers can call OpenDX modules from their own modules or from a separate stand-alone program using the DX CallModule interface. Developers can also control the OpenDX executive from a separate program using the DXLink Developer's toolkit.

OpenDX Builder

Developers can create new OpenDX modules that perform application specific tasks with the aide of the OpenDX Module Builder. The OpenDX Builder uses a graphical user interface to create the necessary module files from user-supplied information. The module builder creates a "template" C-code file, corresponding to the basic framework of a "C" program in which the developer then adds the application specific code.

Visualization Process with OpenDX

Visualizing with OpenDX is a five-step process.

1. Gather, collect, or create the data to be visualized.
2. Investigate the best way to visualize the data. Collect examples. Formulate a visual model representing a "vision" of the data presentation.
3. Prepare the data for the OpenDX data model and work through the details of data import.
4. Design the visual analysis and visual transformations required to achieve the vision, using the tools in OpenDX's visual programming environment.
5. Determine and understand the output requirements, from the viewpoint of someone viewing the output and/or responsible for producing permanent visual artifacts.

Each of these steps is explained in more detail below.

Step 1–Collect Data

The visualization process normally starts with gathering, collecting, or creating data, which can come from various sources and in a variety of forms. The data may be collected from samples taken in the field, generated from a computer simulation, or both. To describe a data set, it is necessary to understand the basic data forms. The interrelationships between data set elements usually determine the form of the data. Examples of different forms follow.

Scattered

In a scattered data set, each component has (at least) a location and a data value, but the locations have no particular connection to each other. Intuitively, a scattered data set is merely a collection of data values attached to points scattered in the same coordinate space. In 2-D space, the location is typically expressed as an <x,y> pair, relative to the 2-D coordinate system. One or more data values can be attached to each point location, representing values measured or estimated at that location. However, other than being collected as points in the same coordinate system, the scattered data are not connected in any fashion. An example of a scattered data set is shown in Figure 2.1

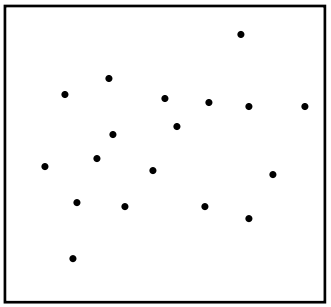


Figure 2.1
Example of Scattered Data

Regular Grid

A regular grid consists of a set of intersection points generated by a set of lines, planes, etc. that are generated in some regular fashion. Such a grid can be defined by an origin point, deltas that define the linear distance in each dimension between grid lines, and counts that define how many grid lines are defined in each dimension. The grid itself connects the intersection points, so that these points are connected in a regular fashion. In one-dimension, a regular grid has point *i* connected to point *i+1*, *i+1* connected to *i+2*, etc. (Figure 2.2). Higher dimensional grids are assumed to use a mesh that is defined in a similar fashion (Figure 2.3).

Figure 2.2
Example of linear regular grid

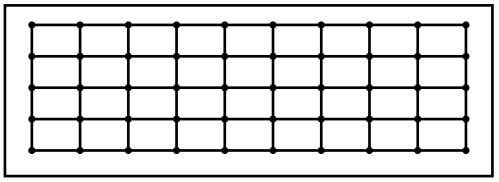


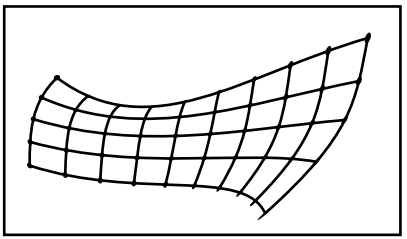
Figure 2.3
Example of a two-dimensional regular grid

Deformed Regular Grid

A deformed regular grid consists of points which are still connected in a regular fashion but in which the set of points is generated according to some scheme, which is not the simple linear relationship defined by constant data values. Figure 2.4 shows an example of a two-dimensional deformed regular grid.

Irregular Grid

An irregular grid consists of a set of points and an explicitly defined set of connections between points. There is no regularity assumed in the points or the connections, so this is similar to scattered points to which explicitly defined connections are added, (Figure 2.5).



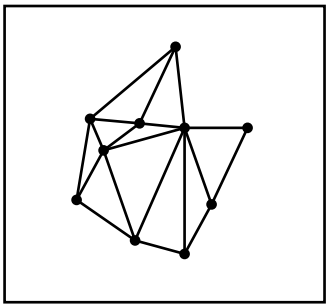


Figure 2.4
Example of a two-dimensional deformed grid

Figure 2.5
Example of a two-dimensional irregular grid

Matching the Data Forms to the Visual Phenomena

For some visual phenomena it is relatively easy to determine which data forms could be used to represent the phenomena. For example, consider the following.

- A regular grid is not an effective way to represent a river, unless the grid size is very small (so that the points could better approximate curves) and the designer doesn't mind having most of the points on the grid represent "not river".
- Scattered data is not appropriate for the typical medical image-scan data set, which is based on a large set of measurements taken at very regular and predictable intervals.

Data Dependency

The issue in data dependency is that of the exact linkage between the location and associated value. That is, is the data value associated with a particular location or a particular space defined by a set of points, i.e. location-centered (Figure 2.6) or cell-centered (Figure 2.7)? In OpenDX, cell-centered values are referenced by a set of connections, though each cell can still be uniquely identified by a "center-point" or the coordinates of some key point. However, there is an important distinction between using a single point convention to refer to a particular space and its values, and assuming that the values are actually associated with or measured at the point location itself. In OpenDX terms, location-centered values are said to be *position-dependent*, whereas space-centered values are said to be *connections-dependent*.

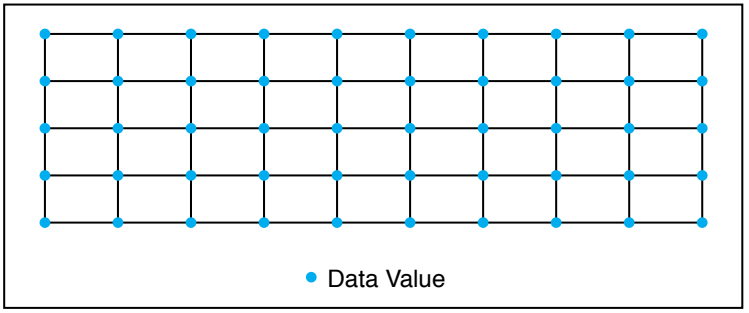


Figure 2.6
Example Position Dependent Data on a Regular Grid

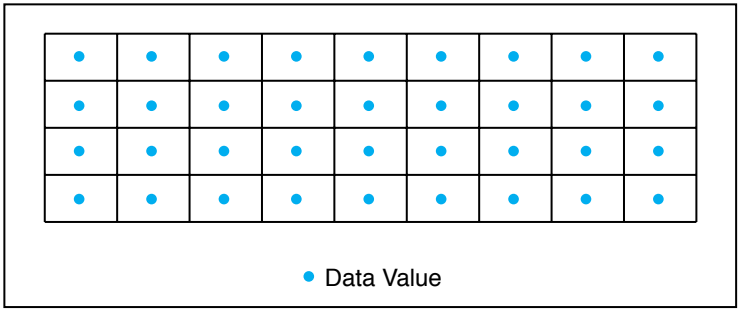


Figure 2.7
Example Cell (Connections) Dependent Data on a Regular Grid

Generally, the decision about which data dependency to use is made when the data set is collected or when the computer simulation is devised. A common way to collect a data set is to use some kind of a grid. For example, a forester may set up a square grid, then count the numbers of tree species inside each grid square or cell. Since the forester collects one set of data per cell, the data are, in OpenDX terms, connections-dependent data. Visually, such data values are assumed to be constant within the grid cell. In this square grid, each grid cell is defined by connecting four corners or vertices on the grid. The positions of the corners are recorded so that the spatial locations of the cells are known. In OpenDX, the paths connecting the vertices are called connections. In this example, the data are dependent on connections and the connections are dependent on positions.

To continue this example, assume that the forester is interested in how the change in elevation over the terrain affects the types and numbers of trees that grow in the study area. Assume that the slope is recorded at each grid point (vertex). A slope data value is valid only at the position at which it is recorded, therefore the slope data set is position-dependent. It is perfectly acceptable to have both position and connection dependent data defined on the same grid. If the forester wants to overlay a continuous set of slope values for the entire grid, slope values can be estimated at non-measured points by interpolating from the known values.

Data Dependency in OpenDX

Data dependency determines how OpenDX treats the data, particularly in relationship to how it assigns data values to locations whose values are not explicitly defined in the data. For position-dependent data, OpenDX interpolates data values between specified positions. For connections-dependent data, OpenDX assumes a constant data value within the space.

The example in Figure 2.8 assigns 50 position-dependent data values to the grid points. The result, with data value represented as color, is the image shown in Figure 2.10 illustrating how OpenDX interpolates values between the grid points. In contrast, the example in Figure 2.9 assigns a single value (an average of the corner point values from the Figure 2.8 data set) as a connection-dependent value to the space named by its four corner points. With connection-dependent data, OpenDX assumes the value is constant for all points in the cell, yielding the image in Figure 2.11.

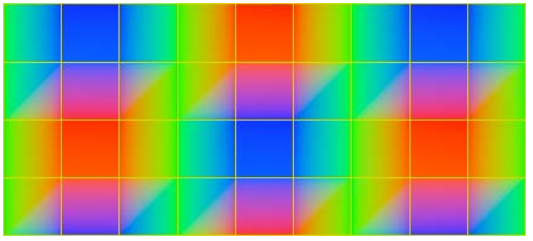
Figure 2.8
An example grid with position dependent data.

0	-0.6	-0.6	0	0.6	0.6	0	-0.6	-0.6	0
0	-0.6	-0.6	0	0.6	0.6	0	-0.6	-0.6	0
0	0.6	0.6	0	-0.6	-0.6	0	0.6	0.6	0
0	0.6	0.6	0	-0.6	-0.6	0	0.6	0.6	0
0	-0.6	-0.6	0	0.6	0.6	0	-0.6	-0.6	0

Figure 2.9
An example grid with connection dependent data

-0.3	-0.6	-0.3	0.3	0.6	0.3	-0.3	-0.6	-0.3
0	0	0	0	0	0	0	0	0
0.3	0.6	0.3	-0.3	-0.6	-0.3	0.3	0.6	0.3
0	0	0	0	0	0	0	0	0

Figure 2.10
Example position dependent grid colored by data value.



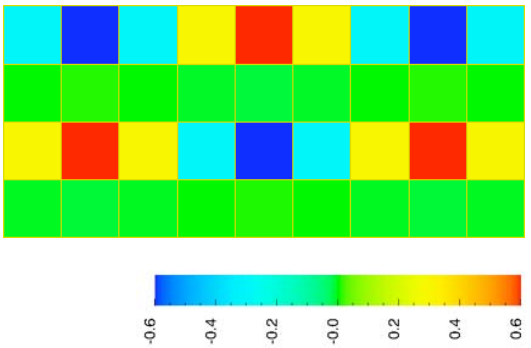
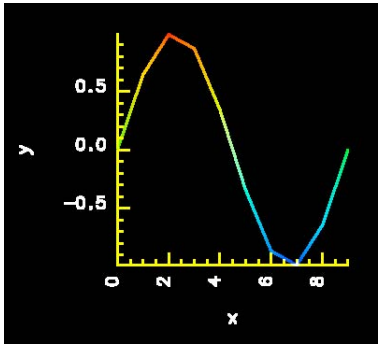


Figure 2.11
Example connection dependent grid colored by data value.

A similar one-dimensional example is depicted in Figure 2.12 and Figure 2.13. In both cases, the location "X" names a 1-D location, whereas the "Y" denotes the data value. Figure 2.12 shows a position-dependent data association, where interpolation between point values produces a line plot. Figure 2.13 shows a connections-dependent data association where the "Y" value is associated to the connection between "X" and "X+1". The result is a columnar plot. In both cases, the data value "Y" is used to add additional interpretation, in the form of color on the line and in the columns, respectively.



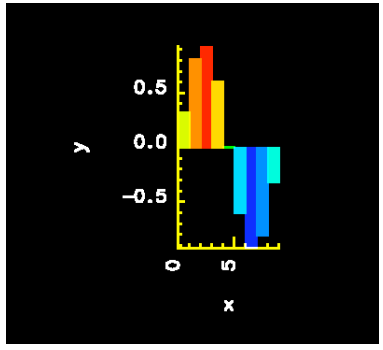


Figure 2.12
Example of one-dimensional data dependent on positions (plotted)

Figure 2.13
Example of one-dimensional data dependent on connections (plotted)

Data Form Helps to Shape the Data "Vision"

The data form and dimensionality help determine what type of visualization is appropriate. If the data form is N-dimensional, a visualization can only show N or fewer dimensions directly. However, the dimensionality can be extended by interpreting data values as values in an extra dimension. As an example of how interpretation can add dimensionality, a one-dimensional, position-dependent data form produces a line when the data values at the points are used to represent values in a second dimension and interpolation is used to provide values between those actually included in the data set. Similarly, a two-dimensional data form can produce a flat colored surface, if the data values are used to represent color or a surface in three dimensions if the data values are used to represent points in the third dimension. By creatively interpreting the data values in the context of the data form, both the data depiction and the data dimensionality can be enhanced.

OpenDX provides a set of "automatic" visualization programs that are used to provide a default visualization for a wide variety of different types of data sets. Many of these defaults for 1-D and 2-D data sets use interpretations that increase the dimensionality as described above. The default visualizations are available in conjunction with OpenDX's point-and-click data import tool.

Step 2–Formulate a Vision

What is the final visualization going to show? How can the visualization be adjusted to show what the researcher wants? These are typical questions that researchers producing visualizations of their data ask. To answer these questions, researchers should:

- gather ideas from published papers;
- look at visualizations that have been done in related fields of study; and
- study the samples provided with the OpenDX visual programming environment.

Gather ideas from several fields of study

OpenDX can be used in a large variety of disciplines. Its facilities are not limited to particular applications, and the same visual effects can often be used effectively across widely different data sets and applications. Thus, looking at a wide set of example applications, such as those in Figure 2.14 through Figure 2.21, may help a user identify an effect that can be effective in a very different application.

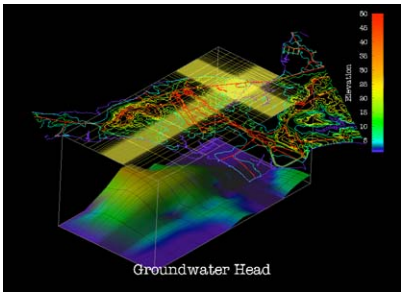


Figure 2.14
Example of Modflow and Arc line coverage; data courtesy USGS.

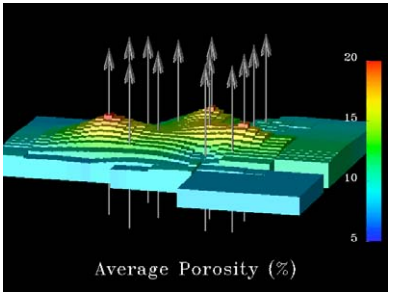


Figure 2.15
Example Petroleum Modeling: Porosity of an Oil Field

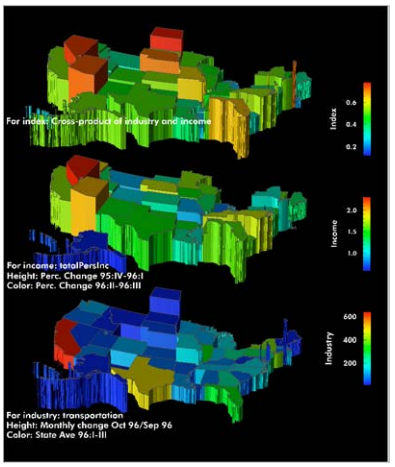


Figure 2.16
Demographic Example
©Chris Pelkie, Conceptual Reality Presentations, Inc.

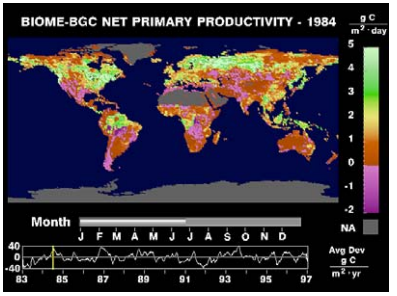
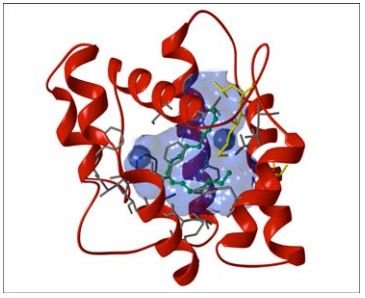


Figure 2.17
Environment Modeling



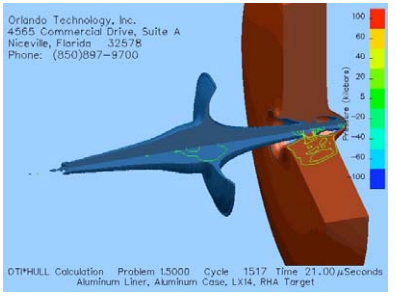
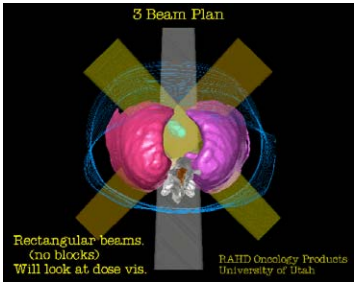


Figure 2.18
Molecular Graphics
Richard Gillilan & Ben Sandler (©Cornell Theory Center)

Figure 2.19
Eulerian hydrocode calculation of a shaped charge jet
©Edwin W. Piburn, Orlando Technology, Inc.



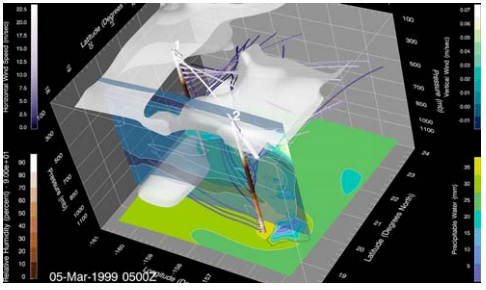


Figure 2.20
Medical Imaging;
©RAHD Oncology Products

Figure 2.21
Weather Modeling
©Lloyd Treinish

Step 3–Importing Data

The most difficult aspect of using any visualization package is getting data into the system. A visualization package that supports only specific applications can make this easier by supporting specific "standard" file formats. Or, a general purpose visualization system such as OpenDX can attempt to minimize the import chore by providing special user-programmable import facilities that are easily adjusted to different data formats. In either case, the data set must be well understood, in terms of the form and dimensionality discussed earlier.

Once the user clearly understands the data form, there are several options available in OpenDX for data import. The researcher can:

- use the General Array Importer to create an appropriate data description (the Data Prompter);
- use the OpenDX ImportSpreadsheet Module;
- use the ReadImage Module to read TIFF, MIFF, GIF, and RGB formats;
- use the OpenDX Import module to read data with a specific structure;
- read directly a OpenDX native format;
- read directly using General Array Importer files;
- read directly NetCDF files;
- read directly CDF files;
- read directly HDF files; and
- read directly CM (colormap) files.

There are several other pre-built data importers available, typically implemented as "filters" that convert data sets in one format into the native OpenDX format. They include the following.

- The system *gis2dx*, which supports data sets in "standard" formats, including DXF, USGS DEM, USGS DLG, ESRI's regular Arc/Info data sets, ESRI's Arc/shapefiles, Oracle data sets, ERDAS LAN and GIS formats, and USGS MODFLOW data sets.
- Multiple specialized format importers have been written by faculty and students at the Cornell Theory Center such as FLUENT, PHOENICS, PLOT3D, SAS, and MODPATH . These can be found at CTC's web address: http://www.tc.cornell.edu/DX.

Beyond these filters, the OpenDX data model is carefully documented, so that most data sets can be massaged, one way or another, into a readable native form.

Importing Data into OpenDX

The interface to OpenDX, Figure 2.22, immediately presents an option to import data. As shown in Figure 2.23, the options for different types of data import are quite extensive.

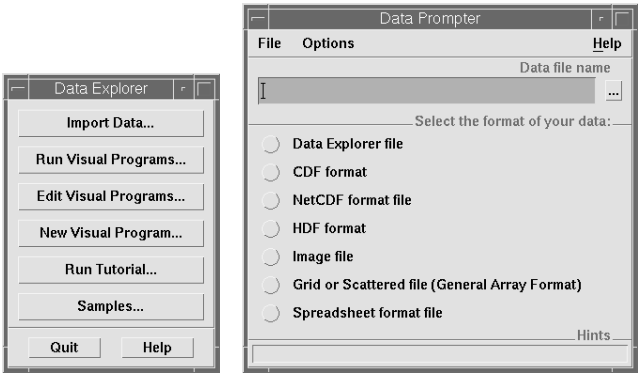


Figure 2.22
OpenDX Startup Interface

Figure 2.23
Import Data Dialog

Many times data are more complicated and must be described with the full data prompter, Figure 2.24.

Figure 2.24
Full Data Prompter

Step 4–Design the Visual Analysis

There are a couple ways to get started creating a visualization once the data set is imported.

Use the Import Data Prompter "Visualize Data..."

The first visualization option is to use the simple default visual analysis built into OpenDX. After importing the data, simply invoke the "Visualize Data" option to tell OpenDX to apply its default visual analysis. It may not produce the image wanted, but this can be a quick and effective way to view the data for initial verification. In some cases this may fail if the default-processing template is unable to accommodate a particular data set.

Create a Visual Program Using the Visual Program Editor (VPE)

The second option is to select the "New Visual Program" option in the Main OpenDX Menu, then use the Visual Program Editor (VPE) to create a new visual program. The VPE's graphical user interface, shown in Figure 2.25, allows the user to rapidly create and modify a visual program that is built from standard OpenDX processing facilities and modules.

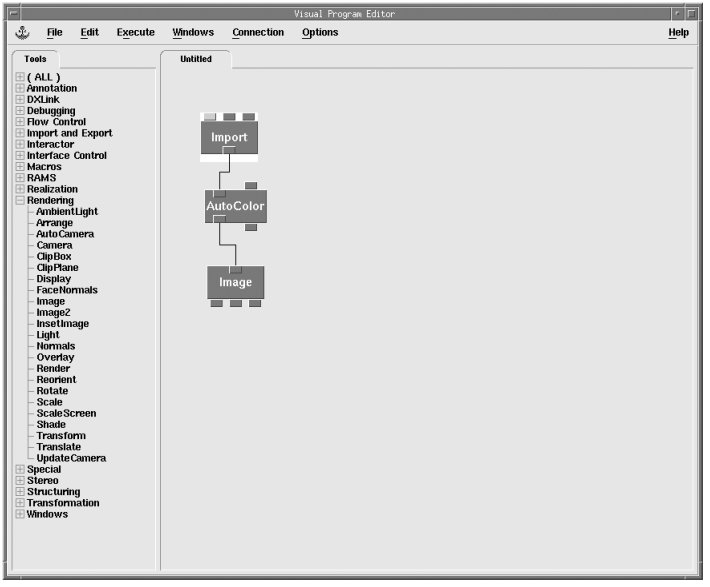


Figure 2.25
Visual Program Interface

Use the OpenDX Sample Programs as a Starting Point

The OpenDX distribution includes an extensive set of examples of OpenDX programs. Included in the "samples" directory. The user can study these examples, then formulate visualization goals and understand how to achieve these goals.

Step 5–Address the visual artifact's output requirements

The desired form of the final output, or *visual artifact*, has a dramatic impact on the way in which the output is derived and exported for storage. Output options include:

- a visual program that allows other users to interactively generate images;
- a single image displayed on an RGB (computer) monitor;
- an animation displayed on an RGB monitor;
- an animation to be converted to NTSC/PAL video signal (i.e., for display on a standard television);
- an image printed on paper; or
- a 3-dimensional object to be displayed and manipulated via VRML.

These different output options have dramatically different image production and display requirements.

- Images on an RGB monitor are limited to the size and resolution of the monitor.
- NTSC Video is limited to 720x486 or 640x480 pixels, and requires 30 frames per second.
- Images to be printed should be higher resolution than that of the image displayed on the computer monitor.

Summary

The process to visualize data with OpenDX is performed using the five visualization steps presented in this chapter. Some of these steps are common to all visualization systems and independent of OpenDX, while other steps are highly dependent on the OpenDX software. This training manual concentrates on those visualization steps specific to OpenDX, so that the user will become proficient in producing the desired visual artifacts. These steps and their relationships to OpenDX are summarized below:

1) Collect data. This initial step is independent from the OpenDX software system. OpenDX uses a general purpose data model capable of describing multidimensional data sets, from simple 1-D data to 4-D (3-D time varying) data to N-dimensional data. It may be helpful to collect data in a format known to be compatible with OpenDX in order to simplify the data importing process. However, OpenDX should be general and powerful enough to handle virtually any data set.

2) Form Vision. Identifying the visualization goal is independent from the OpenDX system. Ideas for the vision can come from various sources like professional journals, Web sites, scientific television programs, and other software packages. OpenDX includes numerous visualization samples from

different disciplines that offer additional visualization ideas, with the added benefit of providing the programs that show how the visualizations are created.

3) Import Data. Importing data into OpenDX and the OpenDX data model are highly system dependent. Several examples in the following chapters will cover all the standard OpenDX data import methods.

4) Define Visual Analysis. Constructing the visual analysis is the most system dependent step. A large portion of this training manual shows how to use many of the OpenDX supplied modules.

5) Prepare Output. Depending on the desired final product, the output can be dependent or independent of the OpenDX system. An interactive, visualization program that allows end users to investigate additional data sets is dependent on OpenDX. However, the visual programmer can incorporate graphical user interface tools supplied with OpenDX to make the final visualization program appear to run independent of OpenDX. Otherwise, OpenDX supports several standard image formats that are software independent. The training materials show how to use OpenDX to export final visual artifacts into all the image formats supported by OpenDX.

In practice, the order of the five steps may vary depending on the problem. Also, the process may involve several iterations, and the vision may change as the visual analysis is refined. After the data are collected, the scientist may decide to import them into OpenDX before identifying a visualization goal. This allows the scientist to confirm that the data can be described and imported into OpenDX before defining specific goals. The scientist can then use a simple visual analysis to preview the data, to gain additional insight into the data and possibly suggest ideas for a visualization goal. An alternative ordering also occurs when the scientist starts with a visualization goal, but no specific data. The scientist must determine how to create or collect the data necessary to satisfy the visualization goal.

The visual analysis part of the visualization process usually involves many trial-and-error data representations and image adjustments. During these iterations, new images are compared to previous images allowing the scientist to determine which image reveals more information about the data. A new visualization goal may result from these comparisons. The iterative process of visual analysis should involve other scientists or team members whenever possible. Other individuals provide additional insight into the data, identify problems with the visualizations, and offer suggestions on how to improve the visualization. These suggestions often mean changing the visualization goal. The first hands-on exercise in the next chapter demonstrates the iterative visualization process where steps 4 and 5 are repeated several times.

First Hands-on Demonstration 3

Using OpenDX

The objective of this chapter is to acquaint you with the user interface, demonstrate how OpenDX creates visualizations, and illustrate some of the most commonly used modules. For this, and all of the hands-on exercises, you are led through the five-step visualization creation process described in the introduction.

Step 1 - Gather, Collect, or Create the data

The data set you use in this lesson is a two-dimensional grid of the earth's elevation, or as it is commonly known, a *digital elevation model* (**DEM**). The particular location of the DEM is in the southeastern portion of the United States. The data values in the DEM are numbers that represent the elevation at each grid point, where positive numbers indicate elevations above sea level and negative numbers indicate locations below sea level.

General information about the data.

- The DEM data format is a standard designated and documented by the United States Geologic Survey (USGS). Elevation data values are located on a regular grid, so that the positions and connections in the data set are both regular. For this particular DEM, the grid origin is (0,0) and the grid increments for the X and Y axes are Delta-x=1, Delta-y=-1. This means that the data set does not have to contain complete lists of grid points and connections–it needs to describe only the elevation values.
- The DEM data file contains elevation values encoded in binary of type "short".
- DEMs store elevation values in column-wise order. (Column-wise versus row-wise will be covered in Chapter 9).
- Finally, the last bit of information about this particular DEM is that the grid size is 301 x 121.

Step 2 - Formulate a Vision

At this point, assume that your vision for depicting the DEM is somewhat uncertain. Generally, assume that the goal is to produce of some sort of map, but whether you want a "flat" (2-D) map (Figure 3.1) or a more realistic looking "landscape" (3-D) map (Figure 3.3) needs to be determined. Thus, the plan is to consider a variety of different visual techniques to display the elevations and then select the one that you like best. You will definitely want to use color to highlight the elevation changes in the data set. The initial visualization will simply color the elevation values. The next will add further details to the map by using contour lines (Figure 3.2), just as USGS topographic maps use contour lines to show lines of equal elevations. The final option will transform the elevation data into a three-dimensional surface, as you might see in a scale-model of a landscape. Examples of the three types of renderings are developed below as illustrated in Figure 3.1 to Figure 3.3.

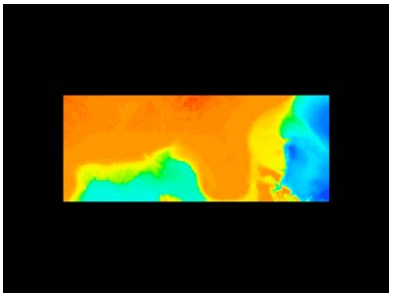


Figure 3.1
2-D (flat) rendering using color to represent elevation.

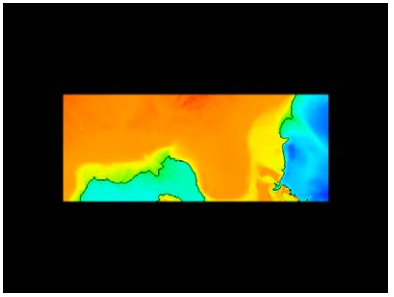


Figure 3.2
2-D rendering using color to represent elevation, adding contour lines to show locations with equal elevations.

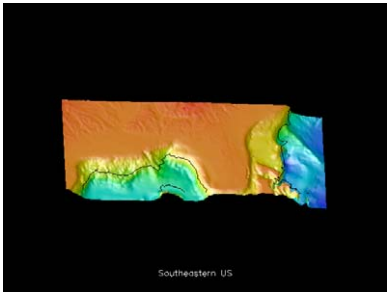


Figure 3.3
3-D (landscape) rendering using elevation as the value in the third dimension, also using color (representing elevation) and contour lines (showing locations with equal elevations) on the 3-D surface.

Step 3 - Importing your data

Now that you have the data and a set of specific visualization goals, you must (somehow) import the data into OpenDX. The objective of this step is to introduce you to one import option: using the General Array Importing facility of the Data Prompter. Once you get the data imported, you can quickly and easily verify the import result using OpenDX's Visualize Data functionality.

Because the data set is organized in a regular and describable manner, you can use OpenDX's Data Prompter and its General Array Importer. After you start the Data Prompter, you simply begin describing the data to OpenDX through its point and click user interface.

- Start OpenDX. Type "dx" at the command line or double click on the "dx icon". The OpenDX startup window appears as shown in Figure 3.4.

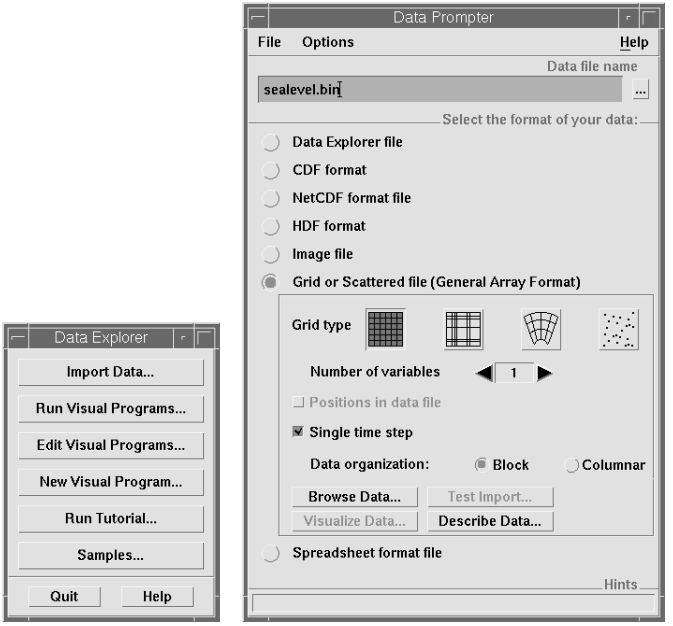


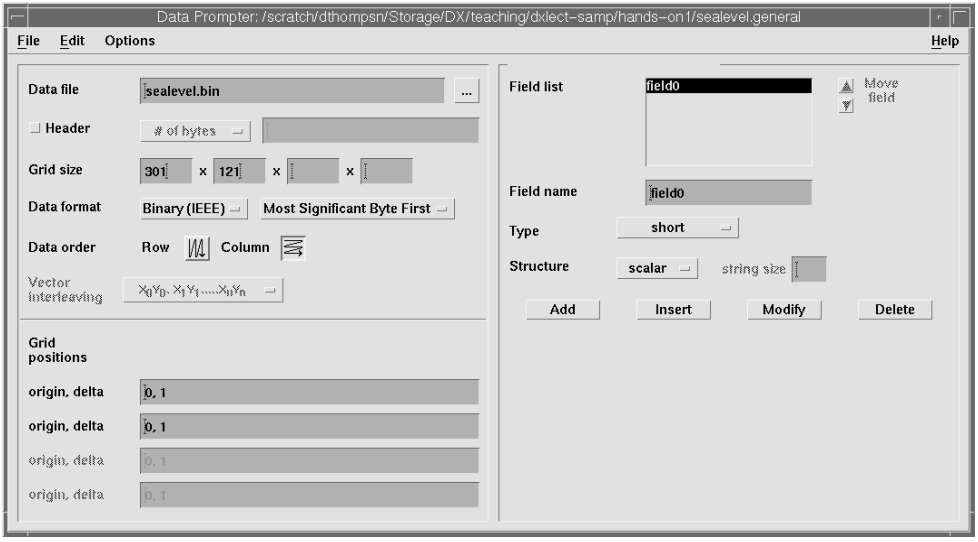
Figure 3.4
Start OpenDX

Figure 3.5
Describing DEMs

- Start the Data Prompter by choosing Import Data from the initial OpenDX window. The Data Prompter, which looks like Figure 3.5, should appear.
- Click Select Data File... under the File menu bar option. Select the data file "sealevel.bin". This file should be included with this material.
- Select the Grid or Scattered file button.
- Select the first Grid type button (representing regular positions, regular connections).
- Click the Describe Data button.
- Begin filling out what you know about this data.
- The grid size is 301x121, so select the first Grid size text box and type "301"; select the second size box and type "121".
- The data are encoded in binary format, so pull down the Data format menu and select "Binary (IEEE)".
- The data are stored in column-wise order, so select the Column button to match this order.

- The type of the first field, “field0” is “short”. Note that field0 is already highlighted, so simply change the “Type” so that it reads “short”, then click the Modify button.
- Save this description produced by the prompter by selecting Save As in the File menu. Name the saved description as “`sealevel`”.

Figure 3.6 *Sealevel description*

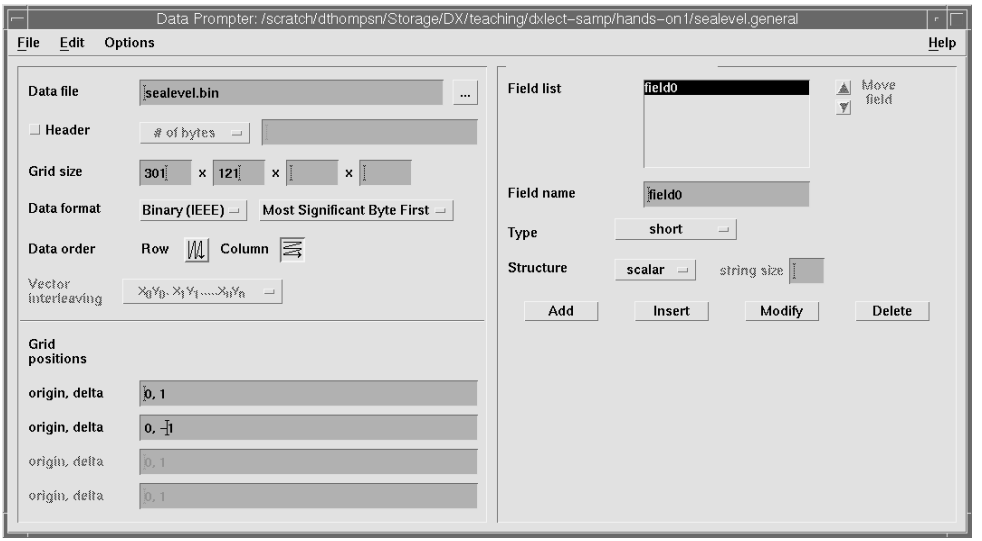


- Use your mouse to move the Describe Data dialog box out of the way, without closing it. Now, click on the Visualize Data button in the initial dialog. Be careful to click on the window's border, not in the initial window's panel. If you happen to click in the panel in the wrong place, parameters may get changed and you will have to start over.

Wait for OpenDX to produce an image. Note that a new set of windows is created, then the Execute menu item turns green until the image appears. Look at the image that results – does it look “correct” to you? Isn't it upside-down from the image shown earlier? What do you think went wrong in your initial data set description? If the elevations are being placed in the wrong (relative) locations, is there some way to adjust the increments on the grid description to fix the problem?

- Go back to the “sealevel” description window. Change the second <origin,delta> pair from <0,1> to <0,-1>. This change says that the Y values go from larger to smaller, rather than smaller to larger as you originally thought. Now, select Save from the File menu to save the modified description.

Figure 3.7 *Modified description*



Try Visualize Data again, and you should see OpenDX produce the image in Figure 3.8.

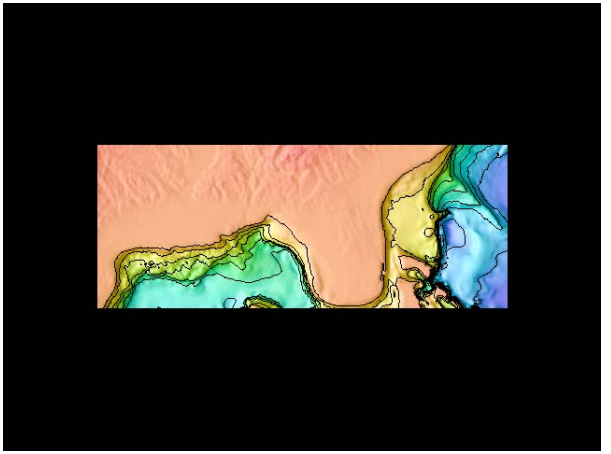


Figure 3.8
Corrected Auto Visualized Image

Evaluation. The default image produced by OpenDX is a nice 2-D map with color, contours, and shading, as you might find on a topographic map. This is very similar to one of your possible "visions", but you should try to use the visual programming environment to write your own program, to see if you can achieve a more ambitious goal.

Step 4.1 - Design the Visual Analysis for Vision 1

Next, you will begin the creation of your first visualization, which involves using the visual program editor (VPE) interface to build a visual program from standard OpenDX modules.

Start the OpenDX Visual Program Editor by choosing New Visual Program from the initial OpenDX window.

OpenDX will open a large, multipart window managed by the VPE (Figure 3.9). A *menu bar* running across the top of the screen contains *pull-down menus* that provide familiar collections of services, such as File, Edit, etc. A large blank area called the *canvas* is where you build the visual program. The two regions on the left are *palettes*, which define a hierarchy of available visualization components, or *modules*. The top palette defines categories of modules, such as Annotation, Realization, or Rendering. The bottom palette lists the modules in the chosen category. The label between the menu bar and the canvas is a *page tab*, which is used to organize larger visual programs that grow too large to fit easily on one screen.

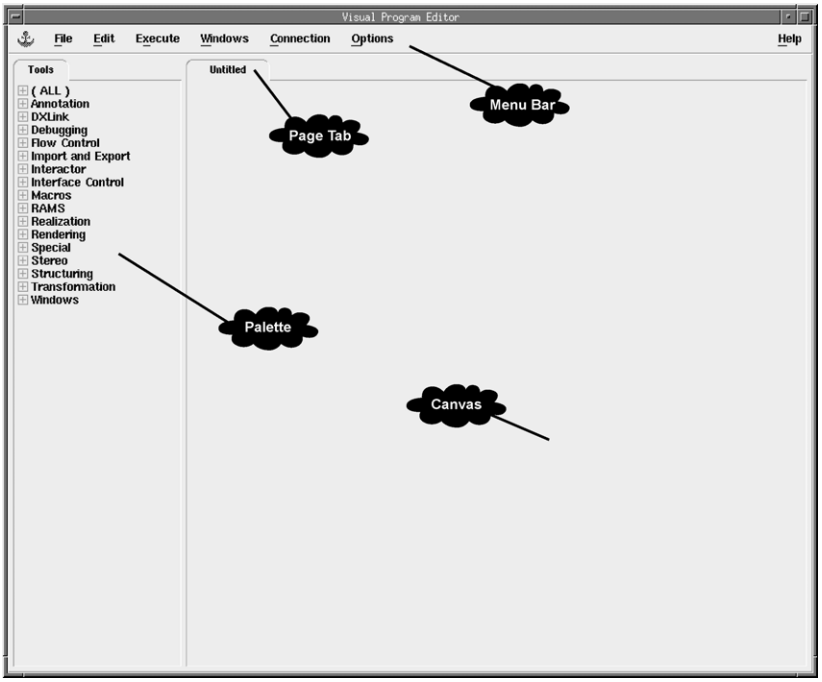


Figure 3.9
The Visual Program Editor

- Start by importing the data file "`sealevel.general`", which you saved in the Data Prompter Import step in the previous section. To do this, click on the **Import and Export** category in the top palette, then click on the **Import** tool in the bottom palette. You should notice that the cursor becomes a corner cursor showing where the upper-left corner of the module will be placed. Move the cursor onto the canvas, position it near the top, then click–this inserts an **Import** module into your visual program.

Note that there are several tabs on the top of the **Import** module icon. Each tab represents a possible input parameter to **Import**. The first input tab of **Import** is cyan colored (the tab circled in red in the Figure 3.10)–this tab color indicates that the corresponding parameter is required. The color of all other parameters indicates that they are either optional or are assigned a default value.

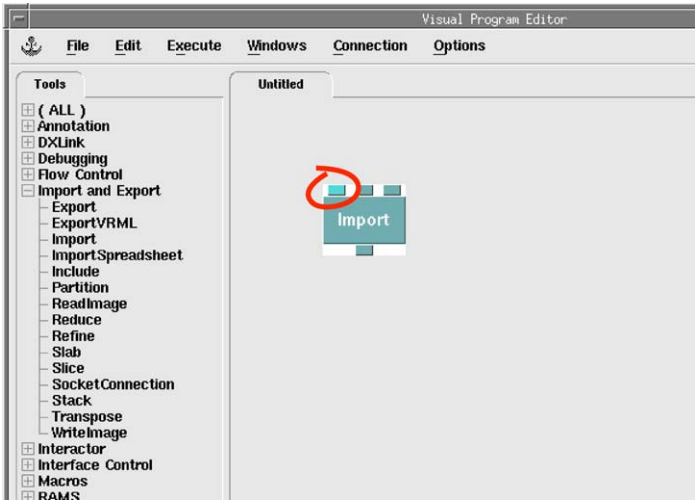
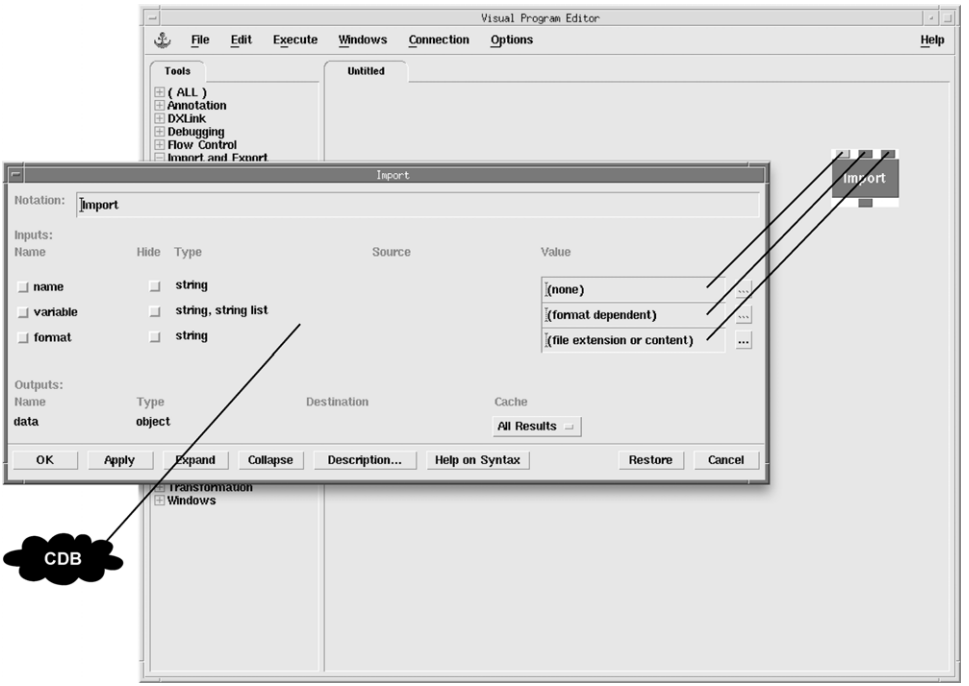


Figure 3.10
A module with required and optional/default parameters

- Open the configuration dialog box (CDB) for **Import** by double clicking on its icon. As shown in Figure 3.11, the three inputs *name*, *variable* and *format* correspond to the three top tabs on the **Import** icon.

Figure 3.11 *Configuration Dialog Box*



- Type the name "sealevel.general" in the *name* parameter box. Next, close the configuration dialog box by clicking OK to record your inputs.

The **Import** module is now displayed with its first tab folded down to indicate that a value for this parameter has now been defined. The tabs that remain up have yet to be defined. Each required parameter must have its value specified prior to execution; an optional parameter will use a default value if no value is specified.

- From the **Transformation** category choose **AutoColor** and place it below **Import**.
- From the **Rendering** category choose **Image** and place it below **AutoColor**.
- Now, use the mouse/left-button to connect the output tab (on the bottom) of **Import** to the left input tab (on the top) of **AutoColor**. You make this connection by positioning the cursor on the output tab, pressing the left-button down, moving the cursor to a position on the input tab, then letting the button up. Connect **AutoColor** and **Image** as shown in Figure 3.12.

By connecting these tabs you direct OpenDX to route the output of one module for use as the input of another. When all modules have inputs appropriately defined, you have a complete visual program. To help you make appropriate connections, you can click on an input or output tab to display the name of that module's parameter.

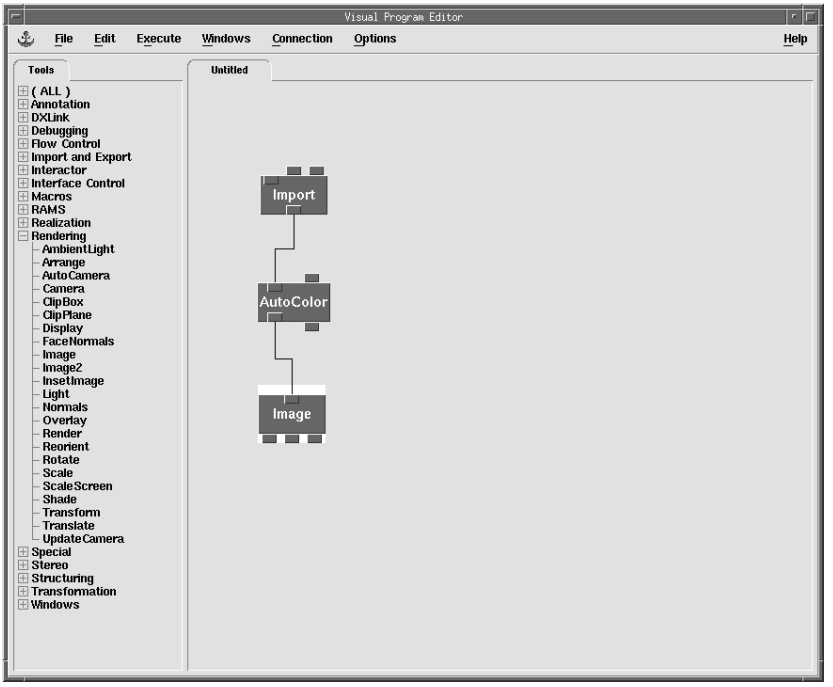


Figure 3.12
First Visual Program

Your visual program is complete, so execute it using the Execute Once choice in the Execute menu. Each module changes color (to green) while it is executing; the Execute option and the page tab also change color while the program on this page is executing. Eventually execution will complete and an image will appear.

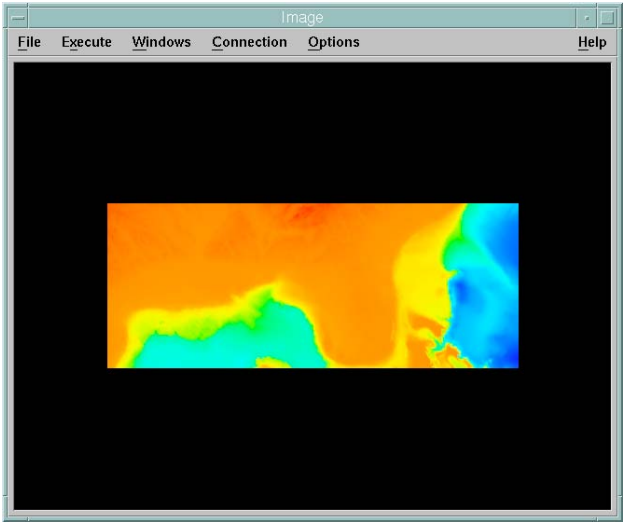


Figure 3.13
First Vision Image

Sometimes execution results in an *execution error*, and no image is produced. If this happens, an error message will appear in a message window. For example, in this program **Import** will return an error if it is unable to locate an input file with the name you specified. If this happens, you need to correct the file name and re-execute the program.

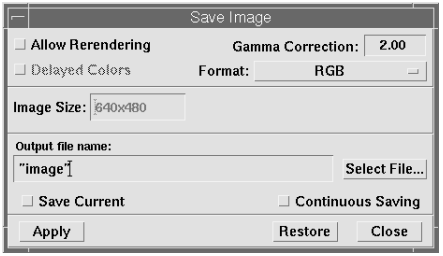
Evaluation. Look at the image. It should match your most basic vision, that of a 2-D map with color used to represent the elevation values. Assume that this is sufficient for now, and move on to consider the final output form you want for this image.

Step 5.1 - Address the output requirements for Vision 1

Let's assume that you want to save this image as a GIF that can be used in a Web page, i.e., as part of an HTML document. Since the image will be displayed on a computer screen, you need to produce the output image to match the resolution required for basic computer display, which is generally 72 or 75 pixels per inch (ppi). Note that images to be printed or used for other purposes may require a significantly finer resolution (a larger ppi).

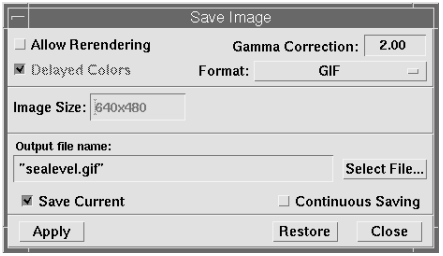
To save your image, click on Save Image... in the File menu on the Image window. The dialog box shown in Figure 3.14 should appear.

Figure 3.14
Save Image File Dialog



To set the output parameters for a monitor display image; change the name of the *Output file name* to "sealevel.gif"; change the *Format* to "GIF"; and click on the Save Current checkbox (Figure 3.15).

Figure 3.15
Save Image File Dialog Filled Out



Note: The GIF format may not be available in your version of OpenDX if the ImageMagick package was not added during compilation. If not, choose another format to save the image.

- To actually save the image using the specified parameters, click the Apply button.

Once the Apply button is clicked, the image is saved and the Save Current check box becomes unchecked.

- Click the Close button to close the dialog box.

The file "sealevel.gif" should now appear in the appropriate directory. You can view the image using any image-viewing package, such as "xv" on UNIX or "PaintShopPro" on Windows, or since it is a GIF image, you can even view the image with an Internet browser.

Step 4.2 - Design the Visual Analysis for Vision 2

Now, revise the visualization and develop a more complex visual program that satisfies more ambitious goals, such as a 2-D map with contour lines. In terms of visual analysis, assume that contour lines are derived from a base elevation map by applying the module **Isosurface** to the base map.

- Return to your program, and modify it by adding an **Isosurface** module, selected from the **Realization** category. You want to include both color and contours, so add a **Collect** module from the **Structuring** category to combine the two into one. Connect inputs and outputs to produce a program like the one in Figure 3.16. Before connecting the **Collect** and **Image** modules, you need to remove the link between the **AutoColor** and **Image** modules. Grab the connection at **Image**, drag it off onto an unused part of the canvas, and drop it there–the connection should disappear.

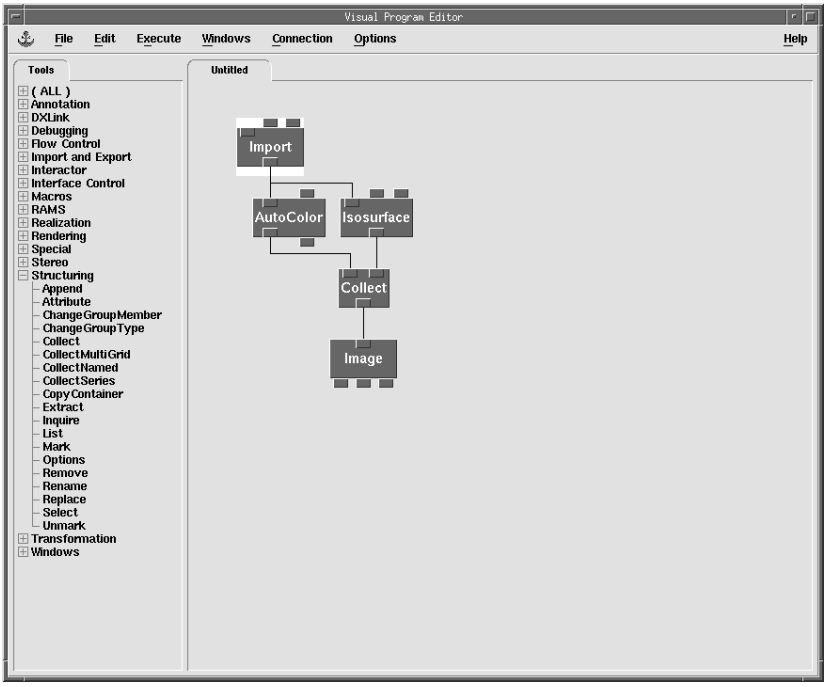


Figure 3.16
Current Visual Program

- Execute the new version of the program.

Look at the resulting image carefully–any contour lines are going to be hard to see, because they are colored yellow by default.

- Change the color of the contour lines by adding a **Color** module (from **Transformation**) between **Isosurface** and **Collect**. You can change the connections between the modules by clicking on the desired **Collect** input tab and moving the link across the canvas to the first input of the **Color** module.
- Open **Color** (double click on its icon) and set its *color* parameter to "black". As shown in Figure 3.17, the "..." button to the far right of the *color* parameter gives a short list of colors, but many others are available. Re-execute the program and look at the new image produced.

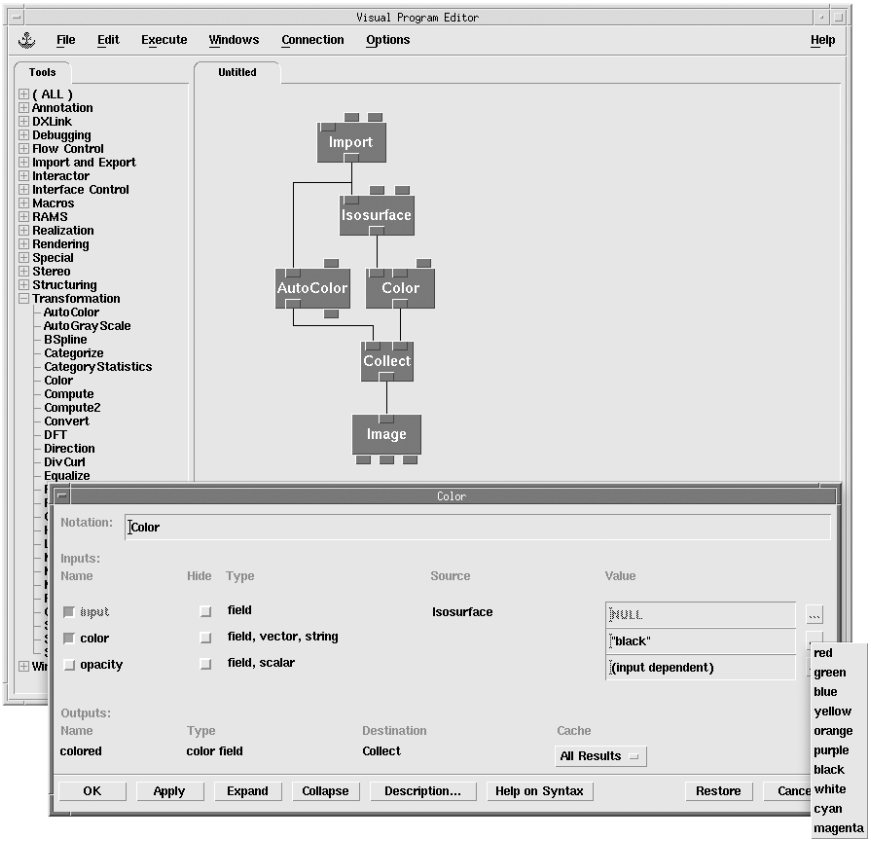


Figure 3.17
The Color configuration dialog box

The output image should now look like the one in Figure 3.18. Does it match your vision? Probably not, because it shows only a single contour line, instead of a set of regular contours. Use the mouse to look at the names of the optional parameters on **Isosurface**. Note that you didn't set the *isovalue* parameter, so OpenDX used a default value. What value do you think was used?

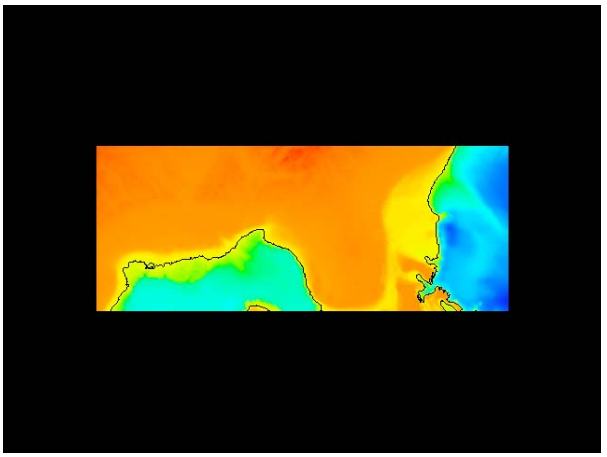


Figure 3.18
Current Image with Contour Line

Open the **Isosurface** module to see what the default is and you should see that it is the "data mean". If you want more information about the **Isosurface** parameters, select the Description button at the bottom of the dialog. Note that there are more input parameters listed than shown in the configuration dialog box. If you return to the configuration dialog box and choose the "Expand" button, OpenDX will show all parameters (some are "hidden" by default).

If you want more information about **Isosurface** you can use the on-line help system. In the "Help" menu, choose "Context-Sensitive Help" and the cursor will turn into a question mark. Now, click on the **Isosurface** tool in the VPE, and a description will appear.

- Open **Isosurface** and change the *value* parameter to 0. Execute again, and the image appears with a single contour line at elevation value "0". This contour line should look like the coastline of southeastern North America.
- Change the *value* parameter to include two values, 0 and 100, by simply typing in the two numbers separated by a space. What do you expect to see in the image? Execute the program to see if you get what you expect.

Control Panels: Changing Parameter Values More Easily

Changing parameter values by typing into configuration dialog boxes is inconvenient, particularly if the visualization designer wants to create a visualization that allows the end user (with no knowledge of OpenDX) to modify parameters. OpenDX provides a facility called an *interactor* that provides a much easier way to interactively change the parameter values of any module. Each interactor is manipulated in a special OpenDX window called a *control panel.* Interactors are particularly convenient when parameter values are to be interactively controlled by a visualization end-user that is different from the visualization programmer.

Before adding a control panel you need to un-set the *value* parameter of **Isosurface**. You do this by opening the module and clicking on the blue square to the left of *value* parameter (Figure 3.19). When you close the module you'll see that the second input tab of **Isosurface** is now up to indicate an undefined parameter.

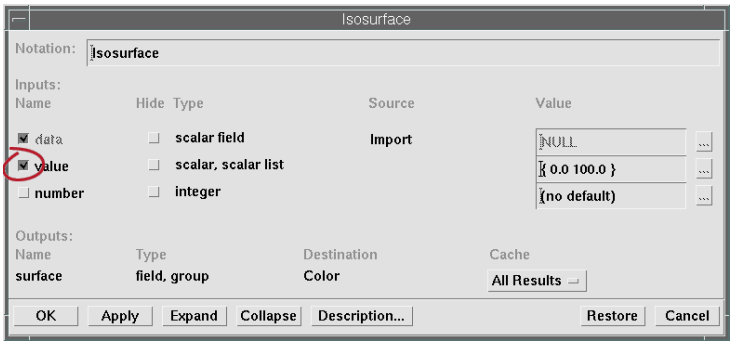


Figure 3.19
Input Set Checkbox

You need to set a scalar contour value, so select the **Scalar** module from the **Interactor** category. Place **Scalar** above **Isosurface**, and connect **Scalar**'s output to **Isosurface**'s *value* input (second input tab). Note that the VPE performs type verification when you attempt to make such connections. For example, when you drag **Scalar**'s output near **Isosurface** all valid connections to **Isosurface** (those for which scalar input values are acceptable) are highlighted green. Figure 3.20 shows how your program should now appear.

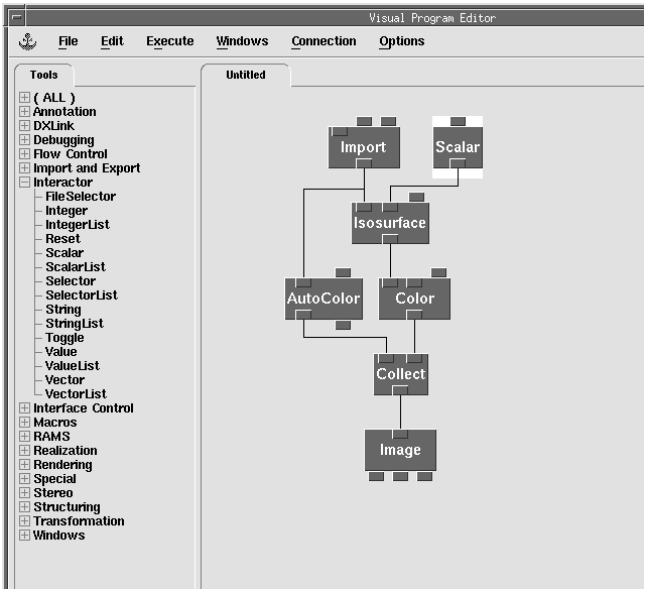


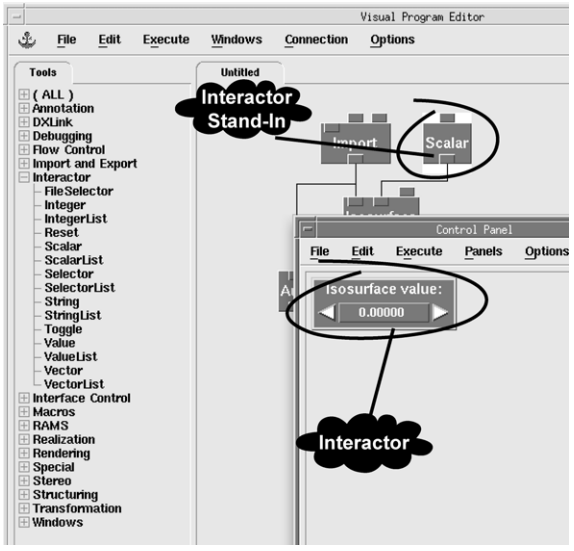
Figure 3.20
Current Visual Program

Now, double-click on **Scalar**–a control panel for the specification of a scalar value will pop up (Figure 3.21).

Figure 3.21
Scalar Control Panel

The **Scalar** module placed on the VPE is called an *interactor standin*. The dialog box within the control panel window that allows the user to actually set the parameter value is called the interactor. Note that the VPE automatically names the interactor as "Isosurface value", based on the module and parameter to which the scalar input is to be sent.

Figure 3.22
Interactor and Stand-in



Setting limits on an interactor

In many cases you need to constrain the user to input parameter values that fall only within a specified range. You do this by setting limits on the corresponding interactor.

- Select the interactor in the control panel by clicking on the interactor's name. Its borders will highlight when it is selected.
- Choose Set Attributes from the Edit menu of the control panel to display the options to set minimum, maximum, and increment values. Set the maximum to 500, minimum to -500, and global increment to 50. (Note: After you type each number, you must press the "Enter" key for the number to be accepted.) Click on OK to record these settings.
- Modify the execution protocol so that any change in a parameter specification results in re-execution. Simply choose Execute on Change from any Execute menu.
- Now use the arrow buttons in the "Isosurface value interactor" to modify this parameter, and see the image result change. After each click on an increment arrow, wait for execution to complete before clicking again. Press down and hold the arrow to increment through a range of values; release when you reach the desired value and the system will execute only with that final value.
- You can always set the value manually, rather than using the increment arrows. Simply click in the value space, type the desired value, and press the Enter key. Do this to set the parameter value back to 0.

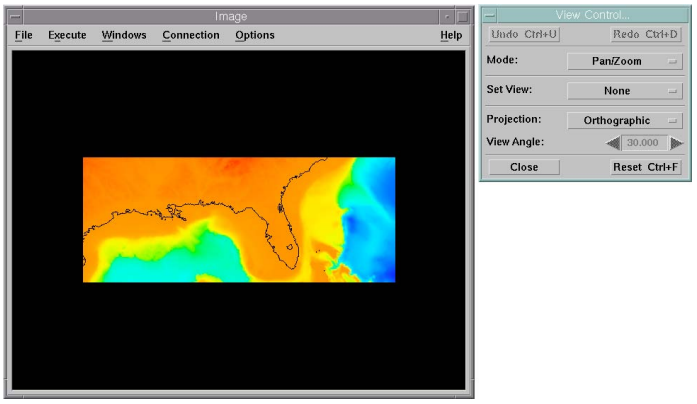
View Control, Colormaps and Data-Driven Interactors

There are several types of interactors that have more complex roles than the simple parameter input interactor discussed above. Many modules have parameters expressed as special interactors to facilitate user manipulation and experimentation with different parameter values. The most important of these are used frequently to set key parameters on **Image** and other modules closely related to image display.

Image provides a wide range of interactive image display options that become available when you activate the Image's *View Control*. For example, you can use the control option to "zoom in" on a part of the image displayed in the Image window. You do this by positioning the cursor in the center of the region to be enlarged, holding the left button down, and dragging away from the center. When you release the mouse button, you define the size and extent of the enlargement. If you simply click on a point, **Image** simply zooms in on a very small region. Zoom out by performing the same actions, but using the right mouse button. After applying these or other view control options, you can always reset the display by selecting Reset from the View Control dialog.

- In the Options menu of the Image window, **choose** View Control. In the View Control window set the Mode selector to "Pan/Zoom."

Figure 3.23
Image View Control



- Use the zoom in technique to enlarge the display of the Florida peninsula. Use the zoom out to return the region to its original size.

Color specification is another type of interaction for which specialized interactive tools are very useful. There are several different color models used in computer coloring that can form the logical basis for interactive coloring.

- Return to the VPE and delete **AutoColor** (highlight the **AutoColor** module by clicking once on its name then press the "Delete" key). Replace this with a **Color** module (from **Transformation**) and a **Colormap** module (from **Special**).
- Connect the modules as shown in Figure 3.24 and execute once. You will see a lot of black areas in the resulting image–what do you think happened?

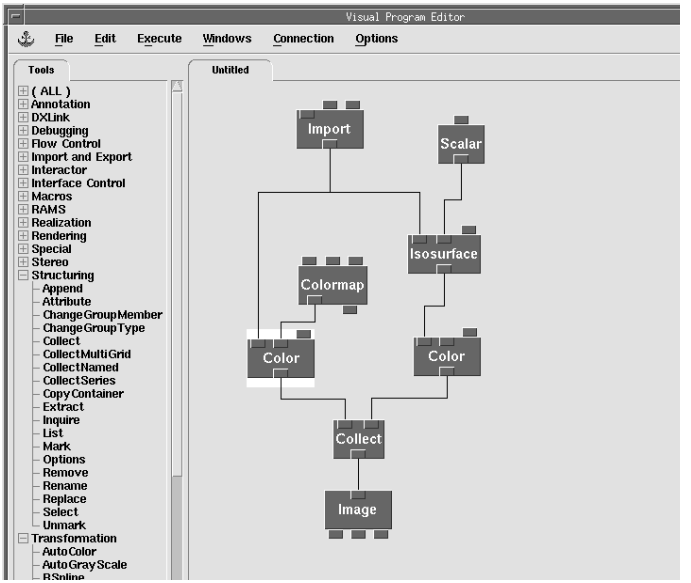


Figure 3.24
Current Visual Program

Open **Colormap** and look at its default parameters–they encompass only data values between 0 and 100. You can reset the limits by hand if you know the range of your data, but it's easier and more general to make the **Colormap** data-driven. You do this by connecting the output of **Import** to the first (*data*) parameter of **Colormap**, as shown in Figure 3.25.

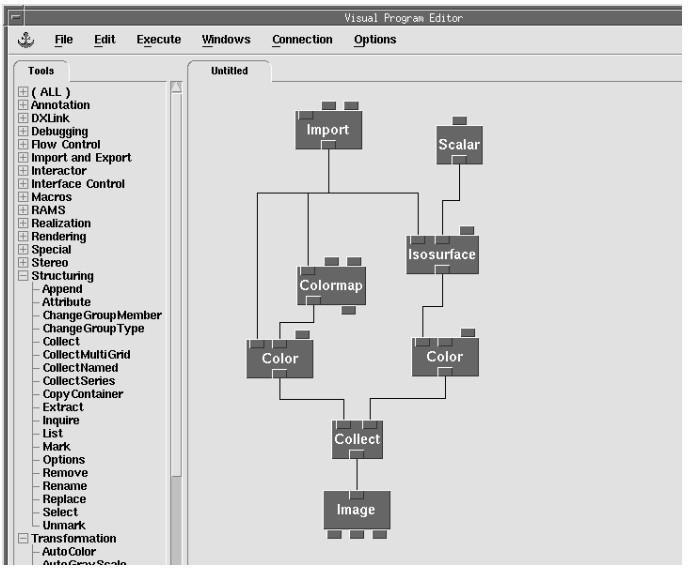


Figure 3.25
Current Visual Program

- Execute again and you should see an image much like what you got using **AutoColor**.
- Now, experiment with **Colormap** parameters using its specialized interactors. You can set control points in hue, saturation, and value space by double-clicking in the appropriate region. Control points can be dragged using the mouse. You can also set control points at precise locations using Add Control Points from the Edit menu.

Other interactors besides those in **Colormap** can also be data-driven. For example, you earlier set the limits on **Scalar** by hand, but you could have simply used the output of **Import** to establish limits on **Scalar**.

- Data-drive the **Scalar** by connecting the output of **Import** to the input of **Scalar**, and execute once.
- Now look at the limits of the **Scalar** interactor in the control panel.

Only one tab on the **Scalar** interactor is shown by default. Other parameters are hidden. For example, you could explicitly specify the minimum, maximum, or parameter increment instead of letting the interactor figure it out. Note that to open the configuration dialog box for a **Scalar**, **Colormap**, **Sequencer**, or **Image** you need to select the module and choose Configuration in the Edit menu.

- Save your program by clicking on Save Program in the File menu of the VPE. Name it "sealevel.net".

Evaluation. The current version satisfies the second vision, that of a colored 2-D map with contour lines. However, you still have yet to achieve the most ambitious vision, that of a 3-D depiction of the landscape.

Step 5.2 - Determine and understand the desired output's requirements for Vision 2.

Assume that you want to save this image as a full-screen image in TIFF format for viewing on a computer screen using high-resolution software. Monitors can typically display a range of resolutions from 800 x 600 up to 1600 x 1200. The amount of VRAM and the computer's video card are the limiting factor for what resolution is displayable. A typical workstation monitor has a resolution of up to 1280 x 1024 pixels. To save your image in appropriate form for this full screen display requirement, do the following.

- Save the image by selecting Save Image... from the File menu on the Image window.
- In the "Save Image" dialog box, change the name of the output file to "sealevel-full.tiff". Change the *Format* to "TIFF". Click on the Allow Rendering checkbox. Change *Image Size* to 1280x1024. Select the Save Current checkbox. Finally, select the Apply button (Figure 3.26).

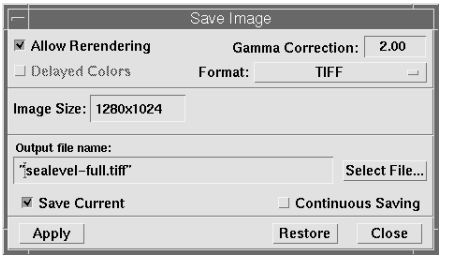


Figure 3.26
Save Image Dialog Box

When you select "Apply", you should notice that the normal cursor turns into a time-cursor, which indicates that OpenDX is currently rendering your image with the new image parameters. Wait for the rendering to finish–you will see the "Save Current" checkbox become unchecked.

Select the Close button to close the dialog box.

The image should now be stored in your directory as "sealevel-full.tiff". You can view this image using any viewing package that supports TIFF format, such as "xv", "PaintShopPro", or "Adobe PhotoShop".

There are important format differences between the GIF image produced earlier and the TIFF image just produced. The GIF format is oriented toward lower resolution display, and attempts to limit file size. Thus, it is limited to 256 colors and uses an LZW compression to compress the data within the output file. TIFF format is oriented toward applications where higher resolution takes higher precedence over small file size, such as electronic representation of color imagery to be included in high resolution printed publications. TIFF supports full 24 bit (16 million) color representation.

Special Note: The ratio between an image's width and height is called its *aspect ratio*. Any time an image of a certain size is displayed or printed as an image of a different size, the image must be adjusted accordingly. A display image that has the same aspect ratio as the source image will generally provide the most faithful representation of the source image; similarly, changing the aspect ratio between source and display generally degrades the quality of the displayed image. A full-screen 1280x1024 display probably does not have exactly the same aspect ratio as the image you were displaying in OpenDX's Image window, so in rendering the output OpenDX will do the best it can to provide you with an image that matches the specified dimensions. In most cases, you can obtain a better result by specifying only one dimension, and letting the OpenDX rendering process select the other dimension to preserve the source image's aspect ratio. For example, you can specify a width of 1280 pixels for "Image Size" but leave the height unspecified; when you render this image, OpenDX performs the calculation to determine the most appropriate height.

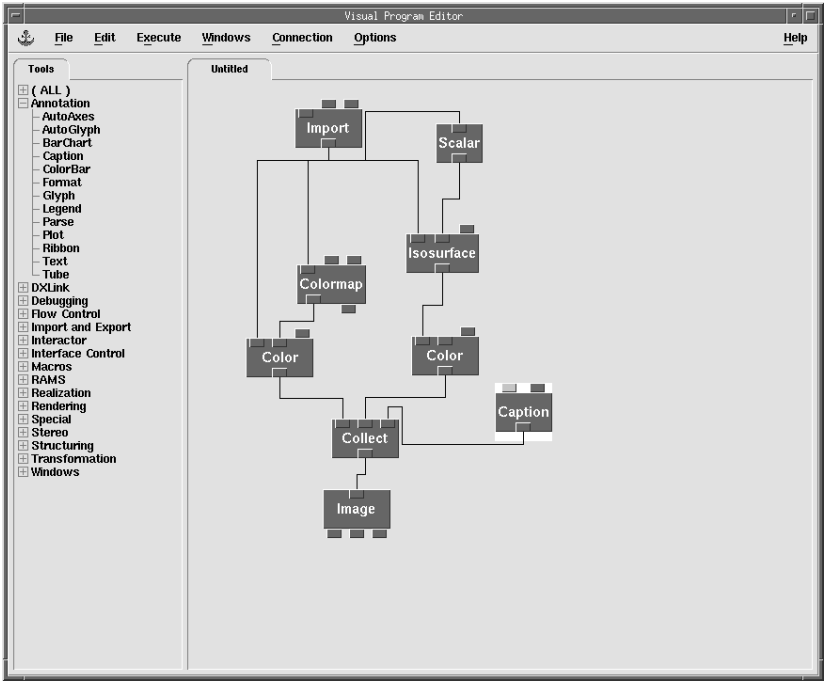
Step 4.3 - Design the Visual Analysis for Vision 3

The goal in the most ambitious vision is to produce a 3-D, landscape-like, depiction of the data set. Thus, you need to apply a visual analysis that produces a 3-D object from a 2-D data form. The idea is to have the first two dimensions defined by the <X,Y> location, and the third dimension defined by the elevation value. In addition, to make the result look truly professional, you might also want to add captions or annotations, and to use "color shading" instead of just plain coloring.

Captions, RubberSheet, and Shading

- You can add a caption to your image by using a **Caption** module (from **Annotation**). The text in the caption will represent one additional layer of information to be combined in the final image, so place **Caption** near **Collect** on the canvas.
- By default, **Collect** has only two input tabs for which both are in use. You will need to add another input tab to allow **Collect** to include the output from **Caption**. Do that by selecting **Collect**, then choosing Input/Output Tabs -> Add Input Tab from the Edit menu of the VPE. Once you've created the new tab, connect **Caption** to **Collect** to give the program shown in Figure 3.27.

Figure 3.27
Current Visual Program



- You define the actual caption by opening **Caption** and typing in a value for the *string* parameter. Other parameters allow you to change the orientation, size, placement, font, etc. of the text. Set *string* to "Southeastern US" and experiment with other parameter values (clicking the Apply button and executing the program to see the result) until you are satisfied with the appearance of your caption.

The critical step in achieving your final visualization goal is to transform the 2-D data set with positions-dependent data into a data set with a 3-D form, where a surface is formed based on logical connections between the points in the third dimension. This process is generally called *rubbersheeting*. The formation of an appropriate surface involves a fairly intricate algorithm, but in OpenDX the process is encapsulated in a single, simple to use **Rubbersheet** module.

- You can form the desired surface from your 2-D data set using the **RubberSheet** module (from **Realization**). Place this module on the canvas between **Color** and **Collect**.
- Disconnect the **Color** with the **Colormap** from **Collect**. Connect **Color** to **RubberSheet** and **RubberSheet** to **Collect** as shown in Figure 3.28, then execute.

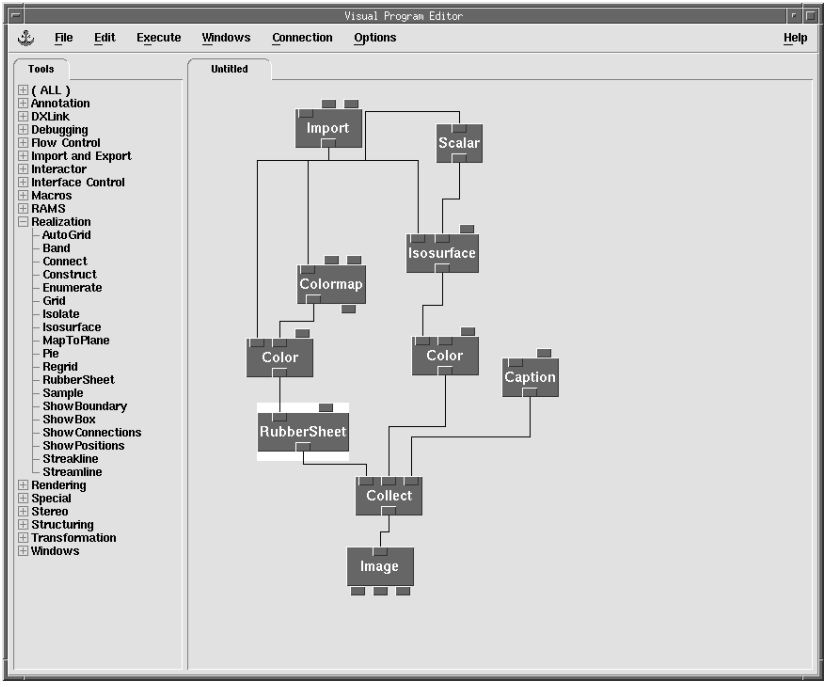


Figure 3.28
Current Visual Program

Your new image should be a three dimensional colored surface. Experiment with additional types of image view control. These are especially useful with 3-D image renderings. For example, you can rotate the image using both left mouse button and right mouse button. It may help if the Rotation globe is visible.

From the Options menu of the Image window, choose View Control, then select *Rotate* mode. To try out various rotation options, try turning on/off the Display Rotation Globe in the Options menu of the Image window.

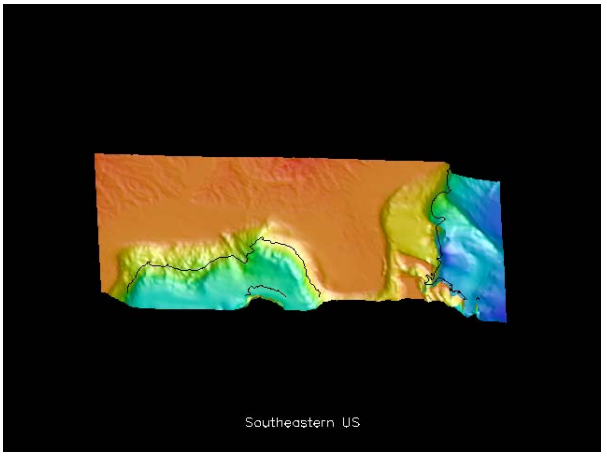


Figure 3.29
One view of the final 3-D image

When you are satisfied with the appearance of the display, save your program.

Evaluation. Your third version of the visual program achieves the third, most ambitious vision for the depiction of the DEM data.

Step 5.3 - Determine and understand the desired output's requirements for Vision 3

Assume that for this image, you want to save the result as a high-resolution postscript file to send to a color laser printer. The image should fill as much of the page as possible without cropping.

Select Save Image... in the File menu on the Image window to open the dialog box. Set the parameters there as follows. Change the *Output file name* to "sealevel.ps". Change the output *Format* to "Color PostScript". Select the Allow Rendering checkbox. Change *Image Size* to "2250x" and press the "Enter" key, see Figure 3.30. Select the Save Current checkbox. Finally, select Apply to render and save the result image.

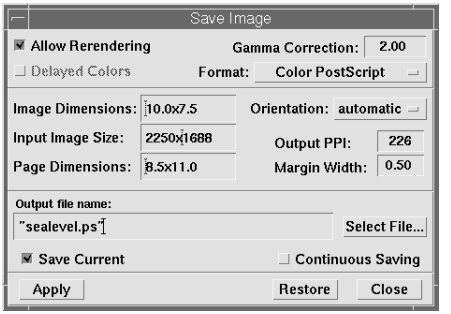


Figure 3.30
Save Image Dialog

When you select "Apply", you'll see the cursor change form to indicate that OpenDX is rendering the image; you may also note that higher resolution output requires a larger and more complex output image, which requires longer rendering time. Wait until the rendering is finished and the "Save Current" checkbox becomes unchecked.

Click the Close button to close the dialog box. The image should now be stored in your directory as "sealevel.ps". You can print this image out to a color laser printer.

As you may have noticed, the Output PPI changed when the large number was entered for the image size. OpenDX will automatically try to create an image that can fit on one page, which may increase the number of pixels per inch. However, it may not always be beneficial to increase the number of pixels being rendered. If the data set is small and the rendered image has no gradient changes, increasing pixel resolution is a waste of time.

First Hands-on: Review

The following is what you should have garnered thus far.

1. You used a particular instance of the OpenDX data model. This instance has a form in two dimensions with regular points and connections, i.e., a regular 2-D grid. The data set is associated with this form in a positions-dependent manner, i.e., corresponding to elevations measured (or estimated) at particular locations in the grid.
2. You were introduced to and gained experience using OpenDX's Visual Program Editor, and should have mastered the visual programming techniques required to select and place modules on the programming canvas, connect/disconnect module inputs and outputs, open modules to specify their parameter values explicitly or select other module-specific facilities, select various menu options, and interact with control panels and interactors.
3. You used a simple, standard data import mechanism to import a simple, regular data set. You also used OpenDX's default visualizer to get a quick, standard view of the data set.

4. You watched OpenDX depict its execution flow as it executes, and selected various execution modes. You saw OpenDX pass data downward from module to module, and noticed that the flow is one-way only–data never flows upwards (except in the special cases of **Get** and **Set** which will be discussed later.)

5. You learned to open a module to access its configuration dialog box, as a way to set parameters. This is a way to set parameters that change infrequently, but is cumbersome for parameters that change frequently–for those, you've learned to use interactors. The dialog box also provides access to special interactive controls on modules such as **Sequencer**, **Image**, **ColorMap**, and other interactor stand-ins, but these modules have to be opened in a special way. You learned a little about some of the controls provided by these special interactors, particularly those on the Image window.

6. You learned that most parameters in most modules have reasonable default values. You also learned how to use simple Interactors and control panels to allow simple inputs to be changed interactively.

7. You learned various ways to save images, with the various save parameters dependent on how you plan to use the output image.

8. Finally, you learned a little about the options provided on the most commonly used OpenDX modules. Most importantly, you learned that in many cases you can use these modules without understanding exactly how they perform their task, what all their possible inputs are, etc. Hopefully, you realize you can evolve your visual programs incrementally, expanding your use of new modules and new options as you expand your vision to more ambitious goals.

Second Hands-on Demonstration 4

Using OpenDX

The objective of this exercise is to further acquaint you with the VPE user interface, explore how OpenDX handles multiple data sets, present several additional commonly used modules, and introduce OpenDX's animation capabilities. As before, in presenting the hands-on exercises, you are lead through the five-step visualization process described in the introduction.

Step 1 - Gather, Collect, or Create the Data

This lesson is based on analysis of complex atmospheric data produced by a computer simulation of a severe thunderstorm. There are three data sets that use the same 3-D grid to position the data. Each data set contains different types of data values, namely air temperature, cloudwater amount, and wind velocity vectors for each grid point. These data sets were provided courtesy of NCSA at the University of Illinois, Urbana-Champaign.

General information about the data:

For each of the three data sets, data values were created for points spaced at regular intervals over a 3-D space. Therefore, it is natural to use a data model in which the data values are associated with the points in a regular grid, with regular positions and connections, and with the grid origin at (0, 0, 0). By looking at the spatial scale of the simulation used to produce the data sets, you see that the appropriate grid size has increments for the X, Y and Z axes such that Delta-*x* = 4166.67, Delta-*y* = 2214.29, Delta-*z* = 4153.85. The grid size in terms of the number of data points is 25 x 8 x 14. Finally, note that in a regular data set such as this, the data set typically does not contain an explicit list of the grid point positions and connections - the positions and connections are generated from a description of the regular structure, using the origin, number of data values, and deltas.

The data associated with each point includes the following:

- temperature data, stored as floating point numbers that represent estimates of temperature in degrees Celsius;

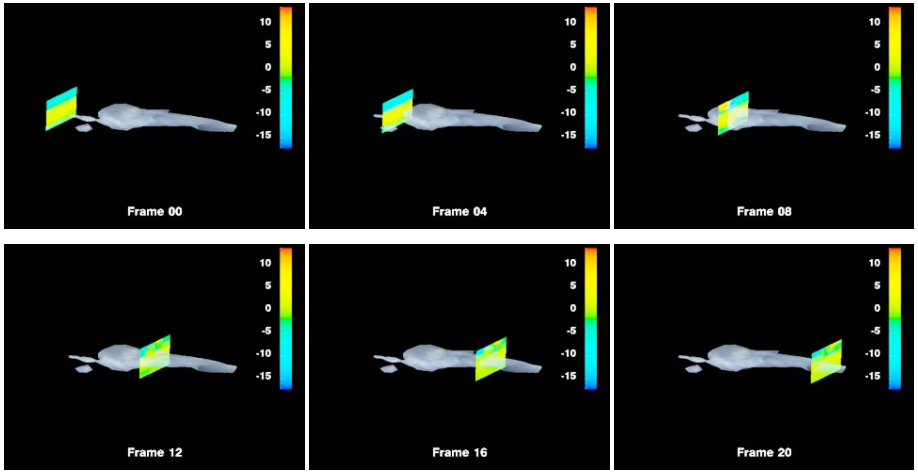
- cloudwater data, stored as floating-point numbers, representing estimates of cloud densities; and
- wind velocity, a vector representing direction and velocity in meters/second, stored as a triple of floating point numbers.

Step 2 - Formulate a Vision

Given these data sets, a meteorologist may know what type of visualization to produce for his/her own specialized interests, but may not know how to present this information to the public. Most members of the public, even those with scientific backgrounds, probably won't know exactly what they'd like to see even if asked. The best approach is often to create some "representative" displays, show them to the target audience, and use the feedback to design more complicated and meaningful images. Thus, the initial vision is to create some representative 3-D image examples to be evaluated by experts and the public. This also allows demonstration and evaluation of various capabilities of OpenDX.

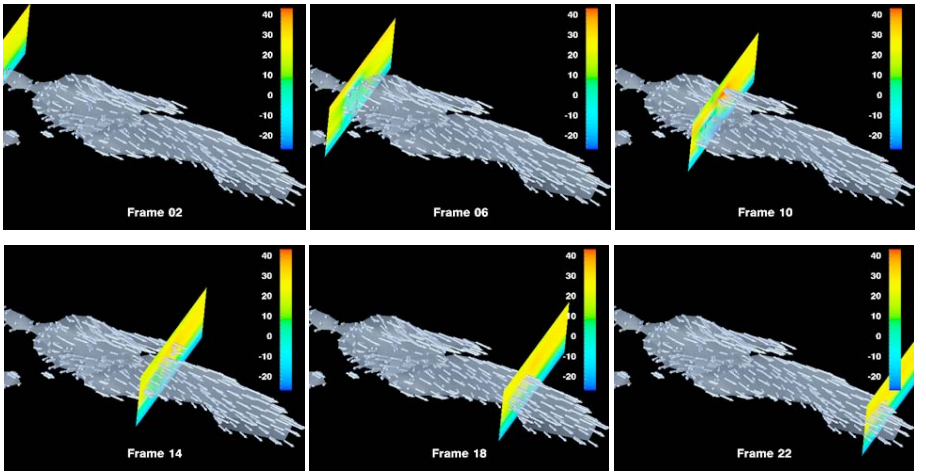
Vision 1 - Show "vertical slices" of temperature through the isosurface of the cloud. Animate this on the screen by moving the slice through the cloud, as shown in Figure 4.1.

Figure 4.1 *Temperature slabs moved through the cloud field.*



Vision 2 - Show a similar sequence of slices of horizontal wind values through the cloud. Show wind as wind vectors with respect to an isosurface of the cloud field, i.e. wind vectors at the points that have the same cloud density value. This is illustrated in Figure 4.2.

Figure 4.2 *Wind data displayed with the cloud field.*



Step 3 - Import the Data

To simplify the exercise, the data have been created and converted to the native Data Explorer format; thus, you can begin creating your visual program in step 4 by simply using the Import module to read the data sets.

Step 4.1 - Design the Visual Analysis for Vision 1

Creating appropriate visual analysis for this application involves using several modules already introduced, along with various new features, The new features include new modules, OpenDX's simple animation techniques, and more complex user interaction features in the VPE interface.

FileSelector Interactors and the Slab module

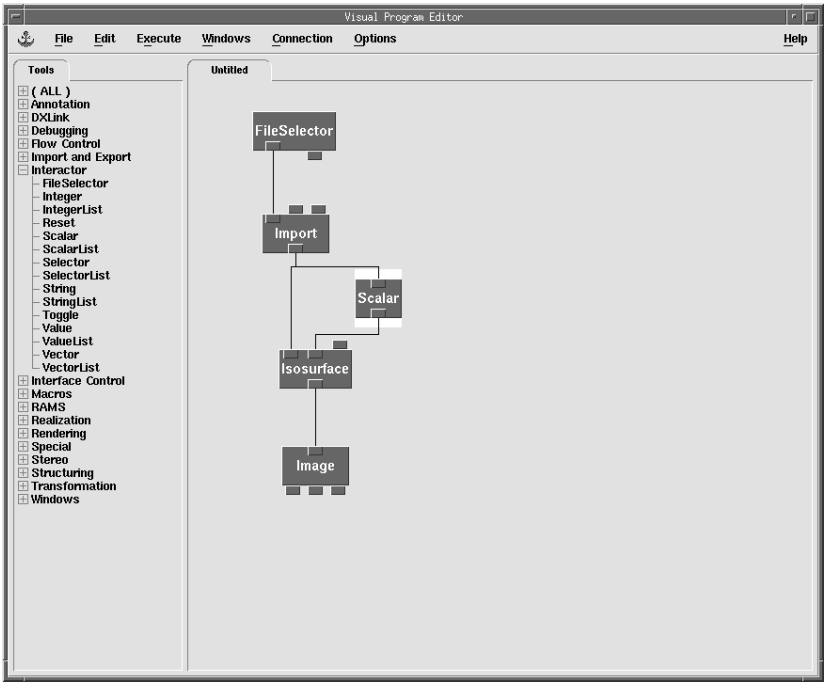
Start a new program in the OpenDX Visual Program Editor by choosing New in the "File menu" of the VPE. [If you are not currently running the VPE, start it directly from the command line by typing "dx -edit".]

Connect an **Import** (from **Import and Export**) to an **Isosurface** (from **Realization**) to an **Image** (from **Rendering**.)

As discussed in Lesson One, interactors allow the end user to change parameters much easier than opening configuration dialog boxes. In this visual program, use the **FileSelector** interactor to allow the user to choose the specific file to import. The **FileSelector** has two outputs: the complete path name to the file, and the file name.

- Place a **FileSelector** interactor (from **Interactors**) onto the canvas. Connect the first output of the **FileSelector** to the first input of the **Import**. Add a data-driven **Scalar** interactor (from **Interactors**) and connect its output to the second input of **Isosurface**, to control the **Isosurface** value. The network should now look like Figure 4.3.

Figure 4.3
Current VPE snapshot.



- Double-click on the **FileSelector** to open its control panel. In the control panel, click on the "…", then navigate and select the data file named "cloudwater.dx" from the samples/data directory of the OpenDX system. Now, execute the visual program, which should produce the image as shown in Figure 4.4.

The image in Figure 4.4 is a rudimentary depiction of the data set. It is useful for confirming that you have selected the right data file and described it correctly, i.e. resulting in something that looks like a "cloud." However, this is obviously not a depiction of the data that provides much insight into the relationships embedded in the data set. Thus, some refinement is obviously needed.

Figure 4.4
Default image for cloudwater isosurface.

- Now add a second **Import** and a second **FileSelector** to the network. You could drag new modules from the appropriate menu, or select those two tools that are already present on the canvas and then *Drag-and-Drop* copy them. To select the two models, either hold down the shift key and click on each module or surround both modules with a rubberband by clicking on the canvas, then drag around the two modules with the left mouse button depressed. After both modules are selected, complete the drag-and-drop copying by using the middle mouse button (both buttons on a PC) to drag copies of the selected tools to a new location.
- Place the new **FileSelector** and the existing **Scalar** interactor to the existing control panel. Open the existing control panel by selecting Control Panel in the Open Control Panel by Name item in the Windows menu. One by one, add the **FileSelector** and **Scalar** to the control panel by dragging (using the middle mouse button or both buttons on two button mice) the module from the canvas into the control panel.

The new Import and FileSelector modules allow the program to import a second data set. Next you want to add to the visualization by taking a 2-D slice of the newly imported (3-D) data set.

- Give the two **FileSelector** interactors names that tell what they do. For example, the first is used for the cloudwater data to create an isosurface, so call it “Isosurface Data”; the second is used to import the temperature data which will be sliced, so call it “Slab Data”. Specify these names by clicking on the interactor's existing name; a white border will appear around the interactor once it is selected. Click Set Label... in the Edit menu of the control panel, then type in the new name. Repeat this for both **FileSelectors** to set both names.
- Change the new **FileSelector** interactor so that it imports the file “temperature.dx”, from the same directory as “cloudwater.dx”. The control panel should look similar to Figure 4.5.

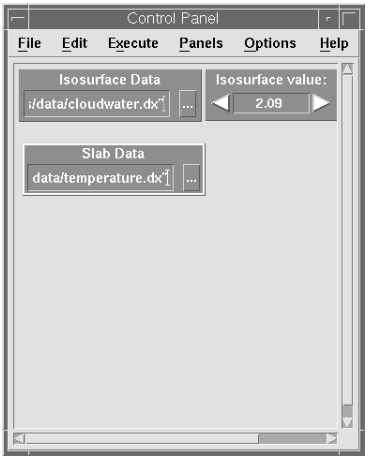


Figure 4.5
Current Control Panel

It is now assumed that you can import all the data, and can use that data to depict simple clouds based on temperature or cloudwater. Next, find a way to slice through the cloud at specified points. The **Slab** module creates a multidimensional object consisting of a selected subset of input data. By using this module, you can slice the three-dimensional data set to form a two-dimensional object, i.e., you can depict data relationships in the 3-D object on a 2-D plane representing the slice.

Add a **Slab** module (from **Import and Export**) below the **Import** that is linked to the "temperature.dx" data set. Connect the output of **Import** to the first parameter of **Slab**. Add an **AutoColor** module and connect as shown in Figure 4.6. Finally, add a **Collect** and connect outputs of **Isosurface** and **AutoColor** to combine both outputs into a single image.

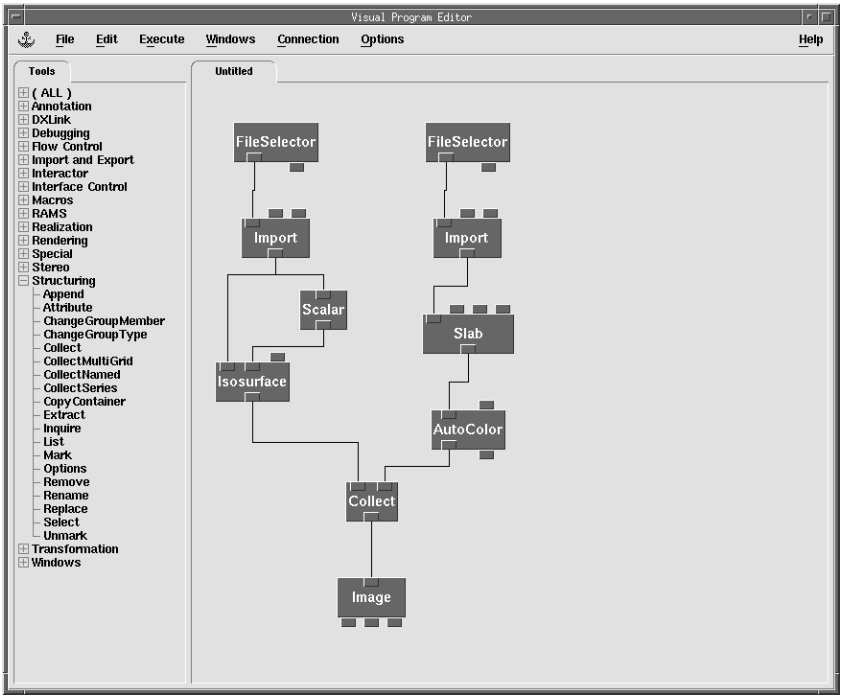


Figure 4.6
Current Visual Program

To control where and how the **Slab** intersects the 3-D data set, you need to set its parameters. Open its configuration dialog box and set the *dimension* parameter to 0, which means that the slice will be taken in the first dimension. Set the *position* parameter to 5, meaning to take the slice in the fifth position in that dimension. The CDB is shown in Figure 4.7.

Figure 4.7
Slab CDB

Using the appropriate interactor in the control panel, set the *isovalue* to 0.1. Now, execute the visual program.

You will probably see only the isosurface, with no slab visible. This is because you are looking at a slab of thickness 0, on edge. Note that a slab in general can be used to capture a 3-D subset of the 3-D data set. However, a plane is by definition a slab of thickness 0. Thus, the plane of interest has been formed, but it isn't visible because of the viewpoint. This is referred to as a slab because OpenDX has another slightly different module called **Slice** (you can use the "help" facility to read about slice.)

- Rotate the object slightly to see the slab. If you don't remember how to perform rotation, please refer back to Chapter 3.

Using the Sequencer to create an animation

- Add a **Sequencer** (from **Special**) and connect its output to the **Slab**'s *position* parameter, resulting in the program shown in Figure 4.8. Remember, to hook this up you will need to unset the parameter first.

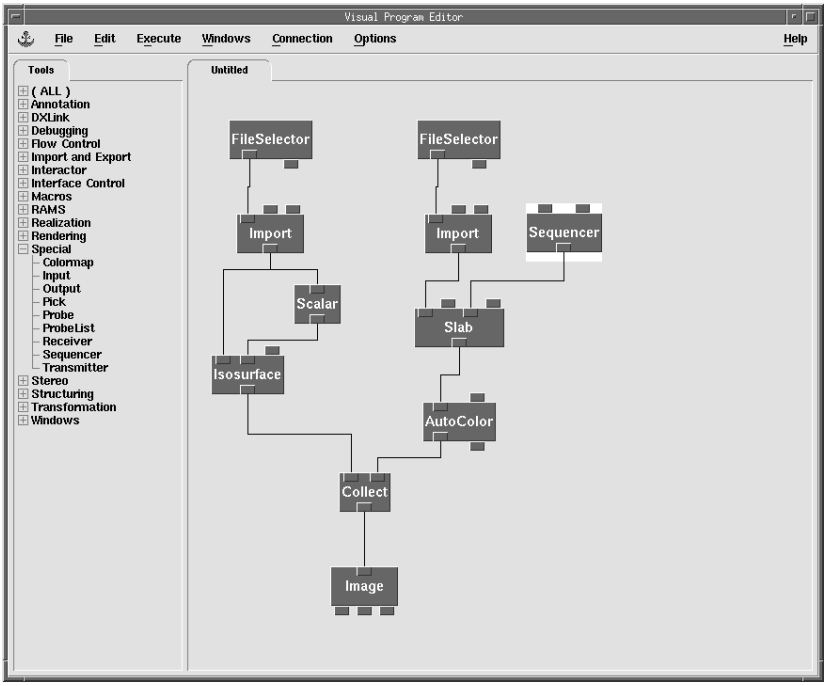


Figure 4.8
Current Visual Program

Consider the impact of wiring the **Sequencer** to the third input of **Slab,** which defines its position parameter. The **Sequencer** outputs a sequence of integers one by one, over a range of numbers easily specified by the user. As the value from the **Sequencer** increases, the slab's position in the 3-D data set changes. Also, the **Sequencer** is system controlled so that each new value it produces represents a "pulse" that pushes through the visual program to produce a new image. Thus, the sequence of values

from the **Sequencer** produces a sequence of similar images, which differ only in the position of the slab. This sequence of images provides an *animation* of the data set.

- Double-click on the **Sequencer** (or choose **Sequencer** from any Execute menu). The **Sequencer** is designed with controls similar to those on a VCR or compact disc player, as illustrated in Figure 4.9.

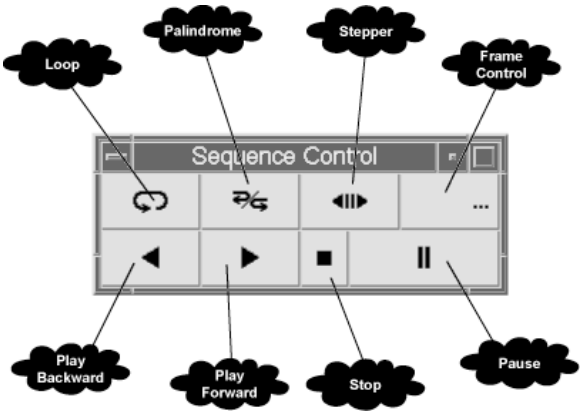


Figure 4.9
Sequencer Interactor

The Play Forward button starts the presentation of the animation by incrementing the data values one by one. The Play Backward button plays the animation by running the sequencer in reverse, i.e., decrementing the data values. The Stop button stops the animation and resets the sequencer so that it will restart with its initial value. The Pause button is similar to the stop button in that it stops the animation; however, it leaves the sequence value set to the last one used. The Loop toggle button causes the sequencer to play from start to end, restart, then play from start to end, indefinitely until the Pause or Stop button is pressed. The Palindrome toggle button causes the sequence to play from beginning to end, then from end to beginning. The Loop and Palindrome buttons can be pressed simultaneously to indicate "continuous palindrome" sequencing. The Stepper button changes the Play from continuous into step-by-step control, in which each Play involves display of only one sequence step. In step-by-step mode, Stepper allows the user to switch to continuous mode. Frame Control opens a dialog box that allows the user to set limits that define values such as the minimum and maximum output values.

- Press the Play button.

A sequence of images should be displayed, but eventually, the system displays a message window noting "**ERROR: Slab:** Bad parameter: position must be an integer between 0 and 24." What has happened? The description of the data says that the grid size of the x-dimension equals 25 but the upper limit defined on the default Sequencer is 100. Thus, you must modify the **Sequencer** so that it outputs integers only between 0 and 24.

- Open the frame control dialog of the **Sequencer** (the top right button), set *Min* to "0" and *Max* to "24". Note that *Max* is 24 (not 25) because OpenDX starts counting at position 0.

As with our earlier examples, once you get a basic image displayed it is helpful to add some sort of annotation to provide context to the image.

- In the Image window, click on AutoAxes... under the Options menu. A dialog box will appear; click on the AutoAxes Enabled button. Close the dialog box and execute once.

Notice with the AutoAxes that you see the dimensions of the data set and can tell the slabs spatial position. There are many options that can be changed with the AutoAxes dialog box and the **AutoAxes** can be added via a module as well.

- Play with the various loop, palindrome, single step buttons on the **Sequencer**.

Note that after the sequence has been completed and displayed once, the next attempt to display the same images seems to execute much more quickly. This is because the OpenDX executive caches the rendered images. That is, the first display requires that the image be computed and displayed, whereas a subsequent display of the same image fetches the pre-rendered image from the cache. This saves rendering time, but because of the size of each rendered image a display of long sequences is VERY memory intensive.

Data driven Sequencer

- Now make the **Sequencer** data-driven. Place an **Inquire** (from **Structuring**) and a **Compute** (from **Transformation**) between **Import** and the **Sequencer**. Wire the network as shown in Figure 4.10.

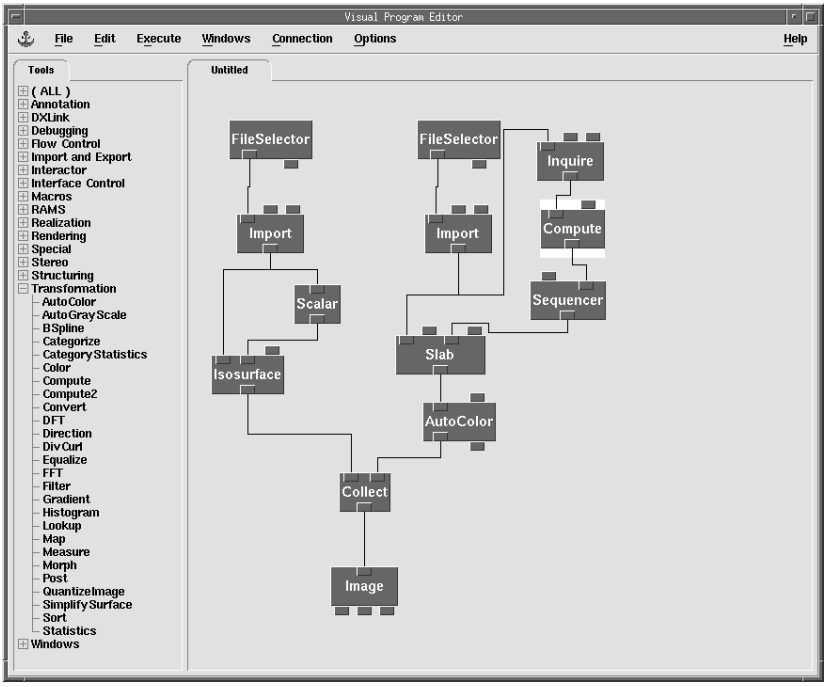


Figure 4.10
Current Visual Program

Set the *inquiry* parameter of **Inquire** to "connection gridcounts" (Figure 4.11). This automatically determines the number of positions, which corresponds to the number of possible slab points in each dimension of the data set. Note that you have to type in "connection gridcounts" as it is not one of the items available in the drop down menu.

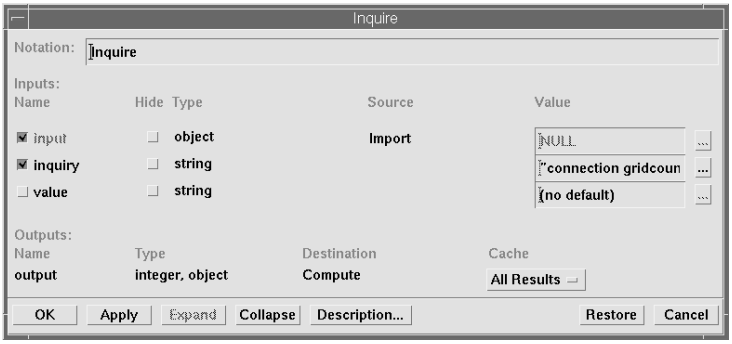


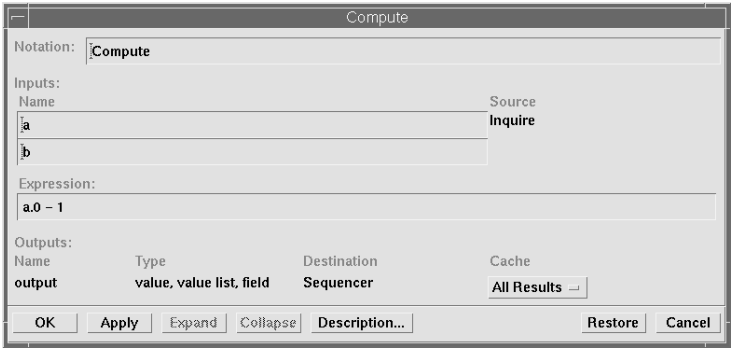
Figure 4.11
Inquire CDB

With this parameter setting, **Inquire** returns a three-vector describing the grid size, i.e. [25 8 14]. You want only the number for the first dimension, i.e. the first element of the 3-vector. To use this to set a maximum value you need to subtract 1, since the slabs are counted starting at 0.

Note that its parameters allow **Inquire** to be used to automatically determine a wide variety of characteristics of a data set. Recall that at any time the full manual page for a module can also be displayed easily. Just select Context-Sensitive Help from the Help menu, which causes the cursor to become a question mark, then click on the module about which help is needed.

- The **Compute** module is used to evaluate expressions. In this case you want to select the first element of a vector "a", written as "a.0" (vector elements are numbered starting at zero) then subtract 1. So, enter “a.0 - 1” as the *expression* parameter of **Compute** (Figure 4.12).

Figure 4.12
Compute CDB



- Select the **Sequencer** tool in the VPE and choose Configuration from the Edit menu. Set the *min* parameter of the **Sequencer** to 0.

ColorBars

Visualizations need to have annotation describing what the information represents. For the current visualization it would be beneficial to know what the colors are representing on the slab.

- Drop a **ColorBar** module (from **Annotation**) between **AutoColor** and **Collect**, and wire it as shown in Figure 4.13. Note that the wire is connected to the **ColorBar** from the second output of **AutoColor**, not the first. Look at the manual page for **AutoColor** to determine why this connection is appropriate.

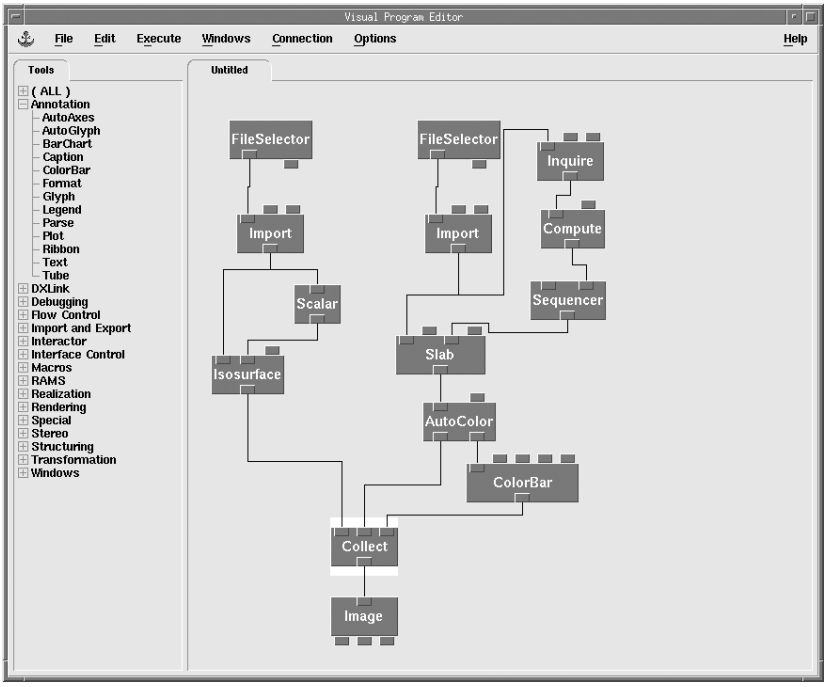
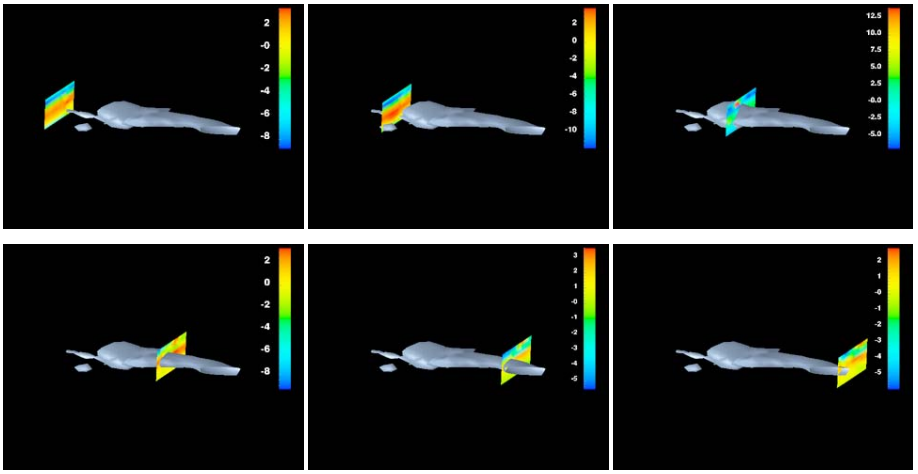


Figure 4.13
Current Visual Program

Execute once. You may wish to turn off the AutoAxes at this point. Select Auto Axes… from the Options menu of the Image window. Click off the AutoAxes enabled button, then click the OK button. Click the Play button.

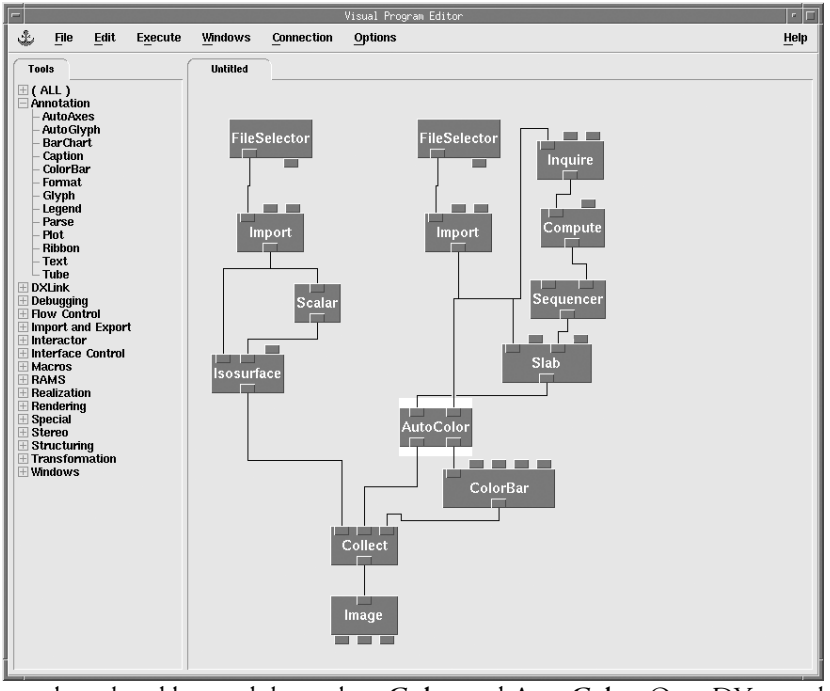
Note that the limits on the color bar keep changing (Figure 4.14). This is because **AutoColor** changes the color scale for each new slab. What would be more useful would be a single color scale used for all slices, i.e., show the range of values for the whole data set, not just those on a particular slab.

Figure 4.14 *Image Sequence*



Tell **AutoColor** to use the entire range of the data to form the color bar by connecting the imported temperature field to the *min* parameter of **AutoColor** (Figure 4.15). This instructs **AutoColor** to find the minimum and maximum of the entire data set, then use that as its range for coloring.

Figure 4.15
Current Visual Program



Just like data values can be colored by modules such as **Color** and **AutoColor**, OpenDX can also change the opacity (transparency) of a color at each data value point. The default opacity for a surface

object, such as an isosurface, is 1.0, meaning completely opaque. In contrast, a value of 0.0 indicates a completely transparent object. For a volume rendered object, the opacity defaults to 0.5, implying half-transparent, half-opaque.

- Change the opacity of the isosurface by inserting a **Color** module between **Isosurface** and **Collect**, and setting the *opacity* to 0.3. This allows the slab to be visible through the isosurface.

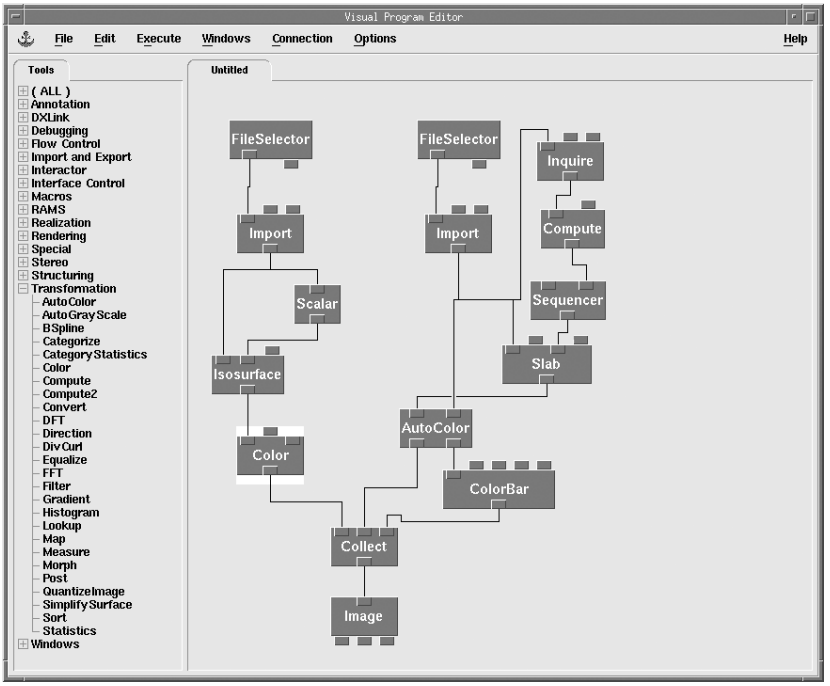


Figure 4.16
cloudwater.net visual program

- Save the network representing the current visual program, by selecting Save Program As… from the File menu, and naming the program "cloudwater.net". Save the program in your home directory for future analysis.

The visual program developed thus far satisfies our first vision for depiction of the data, i.e. an animation displayed on the screen that depicts basic values and relationships in the data sets. This depiction is sufficiently well developed to present to a target audience to get meaningful feedback on what other features or animations might be useful.

Step 4.2 - Design the Visual Analysis for Vision 2

The purpose of the next exercise is to attain a more complex vision for data depiction: creating an animation of a horizontal wind velocity slab, and saving the result to an external file for off-line display.

To attain this vision, we introduce aspects of OpenDX's data interoperability and more user interface techniques, such as pages and annotation.

- Modify the existing visual program to apply the Slab Data **FileSelector** to the file "wind.dx" instead of "temperature.dx", i.e., open the Control Panel and change the name. Note that you can overtype the name "temperature.dx" with the name "wind.dx", but make sure to press the “Enter” key when finished to make this change effective. Execute the program.

When the modified program is executed, all of the modules continue to work correctly, even though you have changed the data set being analyzed from a single scalar temperature value per position, to a vector describing wind. The result is that the colors and structure now represent magnitude of wind velocity instead of relative temperature.

Suppose the user is interested not in the simple magnitude of wind velocity, but rather in the value of vertical wind velocity. To change the analysis to produce this result, the visual program will need to use the **Compute** module used earlier. The three vector that makes up the wind is ordered [x-axis horizontal, *y*-axis vertical, *z*-axis horizontal], so the expression to select the x-axis horizontal wind velocity is "a.0". The expression to select the vertical wind is "a.1".

- Place a **Compute** between the **Import** and the **Slab**. Make sure you connect the output of **Compute** to the *min* parameter of **AutoColor**, Figure 4.17. Define the *expression* in the new **Compute** as "a.1", to select the vertical wind component, and run the **Sequencer**. Next, change the *expression* in the **Compute** to "a.0", to select the x-axis horizontal wind component, and run the **Sequencer**.

If you look at the results of the visual analysis, note that the various data depictions begin to reveal characteristics of the data that are not obvious from just looking at a collection of numbers. For example, if you compare the versions that show horizontal versus vertical wind, you see that the highest vertical winds are strongly concentrated in the center of the storm, in contrast to the horizontal winds, which are more constant throughout the data set. Thus, selecting the right depiction is potentially valuable in finding and illustrating relationships that are otherwise hard to deduce without the proper display.

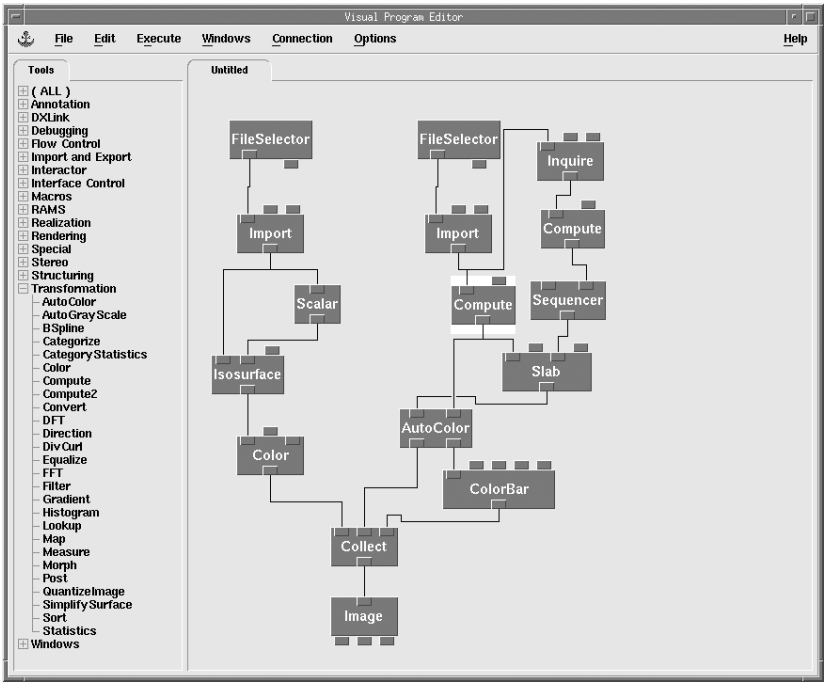


Figure 4.17
Current Visual Program

Pages and Annotation

By now, you have added so many modules to your program that the canvas is crowded and making the visual program somewhat hard to understand. It is time to look at features that can be used to organize visual programs as they get larger and larger.

Currently there are three different objects feeding into the **Collect** before **Image**. Place a **Transmitter** (from **Special**) onto the canvas under the **Color** module. Remove the connection that runs from the **Color** to the **Collect**, and instead connect **Color** to the transmitter. Open the CDB for the **Transmitter** and change the *Notation* parameter to "Cloud". Click Apply to record the change (Figure 4.18).

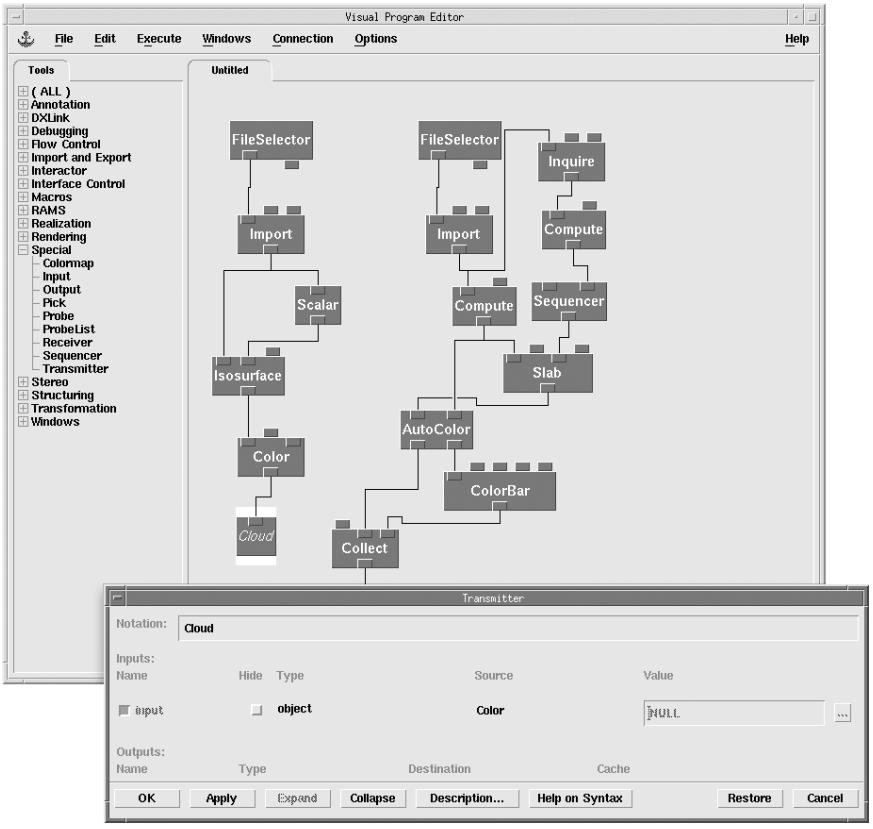


Figure 4.18
Current Visual Program

Close the **Transmitter** dialog box. Place a **Receiver** (from **Special**) above the **Collect**. Note how the **Receiver** automatically names itself the same as the previous selected **Transmitter**. Wire the **Receiver** into the **Collect** and execute.

The network should work the same as it did before the insertion of the **Transmitter**/**Receiver** pair. Using pairs such as these allows the visual programmer to reduce the amount of wires in a network, and to lay out the network in a simpler manner. A named **Transmitter** always sends its data to a **Receiver** with the same name. The names of **Transmitters** and **Receivers** are restricted to letters, numbers, or the character "_" and must begin with a letter. **Receivers** are renamed by the notation field just like **Transmitters**; however, they must have a corresponding **Transmitter**.

Use two more **Transmitter**/**Receiver** pairs to disconnect the **Collect** from the rest of the program. Name the pairs "slab_data" and "colorbar", as shown in Figure 4.19.

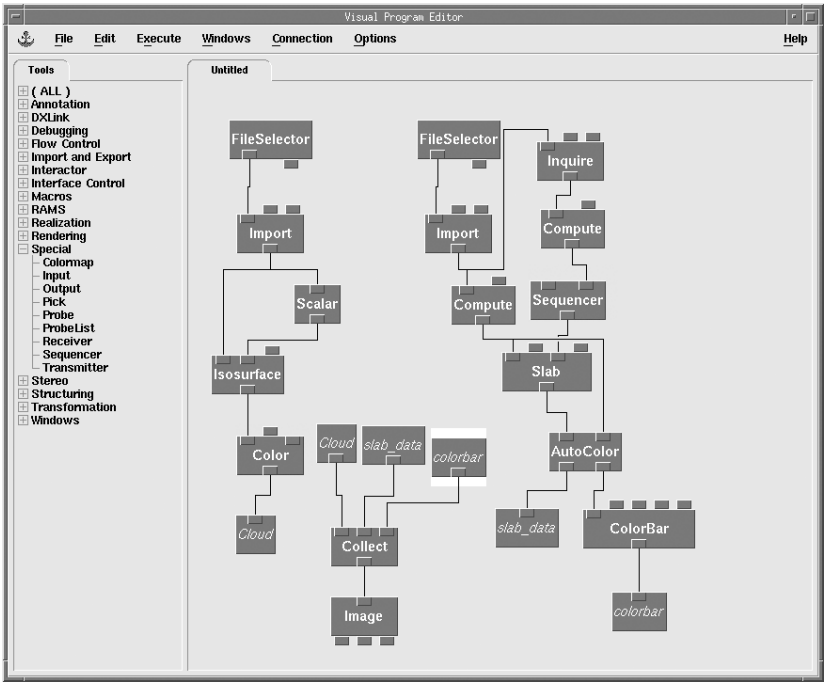


Figure 4.19
Current Visual Program

Note that although this actually increases the number of modules on the canvas, it does reduce "line clutter". More importantly, it allows the use of an OpenDX feature called a Page that allows collections of modules to be encapsulated more cleanly. Pages always communicate with other pages using transmitters and receivers.

- Make a new page for the collection of modules including the **Receivers**, **Collect**, and **Image** modules. Select one of the **Receivers**, then in the Edit menu choose Select/Deselect Tools -> Select Connected to highlight all modules connected to that module. From the Edit menu choose Pages -> Create with selected tools and a new page tab labeled “Untitled_1” appears, while the selected modules disappear.
- Make a separate page for the set of tools that create the cloud.
- Give the pages reasonable names. You can do this using the Edit menu Page->Configure Page option, or double clicking on the page name in a tab.
- Add notes anywhere on the program’s canvas by choosing Add Annotation from the Edit menu of the VPE. The network should now look similar to Figure 4.20.
- Save the program.

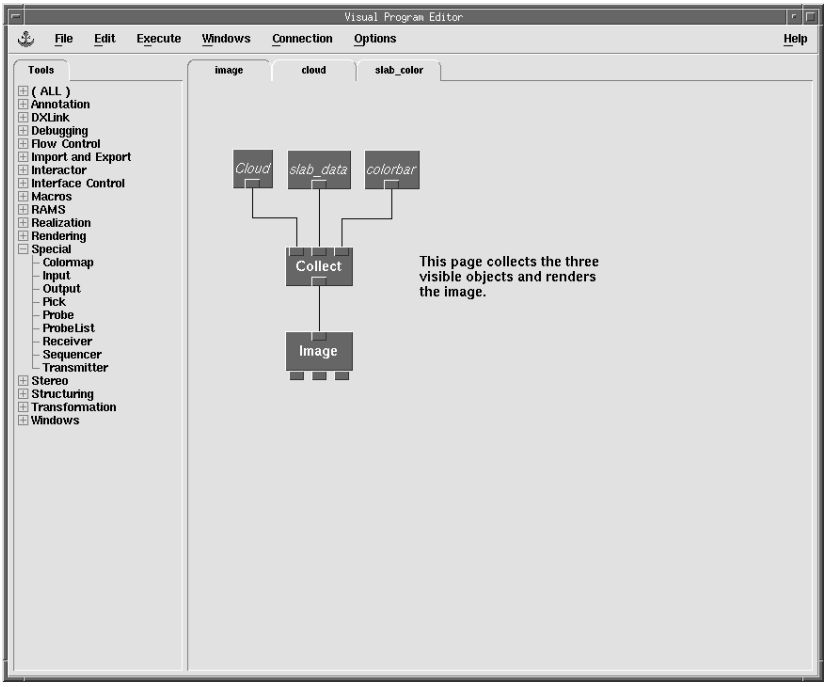


Figure 4.20
Current Visual Program

Glyphs

A *glyph* is an explicit object that visually represents a data value. Example glyphs include arrows that show the direction of vector data, a sphere that depicts the sizes of scalar values, or even something as complex as an airplane that represents a vector moving through a vector field. The key is that the glyph be scaled in size, positioned in direction, or in some other way be adjustable in form to depict one value from a range of possible values. To illustrate, an obvious use of glyphs is as arrows that show wind direction. You will use this with your example data set, focusing specifically on wind direction at the surface of the cloud. In terms of the analysis, the wind data must first be mapped onto the cloud water density isosurface, then arrows can be used as glyphs to represent the magnitude and direction of the wind data.

- Place a **Cloud** receiver in the upper left-hand corner of the canvas on the slab_clrbar page. The easiest way to do this is to first locate the **Cloud** transmitter on the cloud page, select the transmitter, go to the slab_clrbar page, and drop a **Receiver** (from **Special**) on the canvas. By selecting the appropriate transmitter first, the receiver is always properly named for you.
- Place a **Map** module (from **Transformation**) below the **Cloud** receiver. Wire **Cloud** into the first input of **Map**. Wire the **Import** into the second **Map** input.
- Place an **AutoGlyph** module on the canvas and wire **Map**'s output into the first input of this module.

- Place a **Color** module on the canvas, wire **AutoGlyph**'s output into the first input of this **Color** module and change the *opacity* parameter to 1.0.
- Create a new transmitter to send these glyphs to the image page where they will be collected with other parts of the image. Name this Transmitter "glyphs", as shown in Figure 4.21.

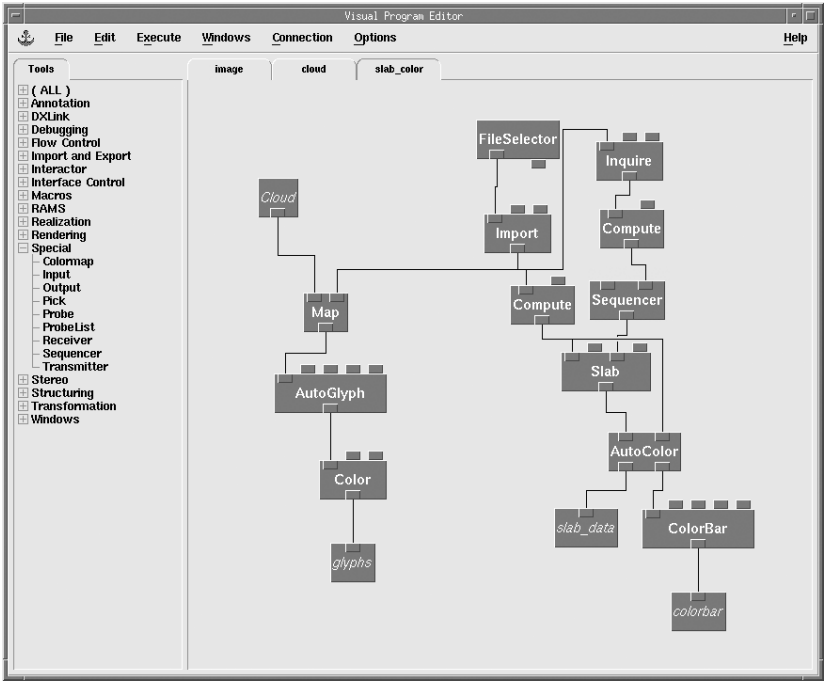


Figure 4.21
Current Visual program

Note that you need to explicitly set the Opacity to 1.0. In this case it does not automatically default to this value, because the cloud has its opacity set to 0.3 from earlier in the network. When the new data are mapped onto the surface, the colors remain the same.

- On the Image page, add another tab to **Collect** and wire the **glyphs** receiver to it. Execute. Zoom in to see more detail (Figure 4.22). Save the network.

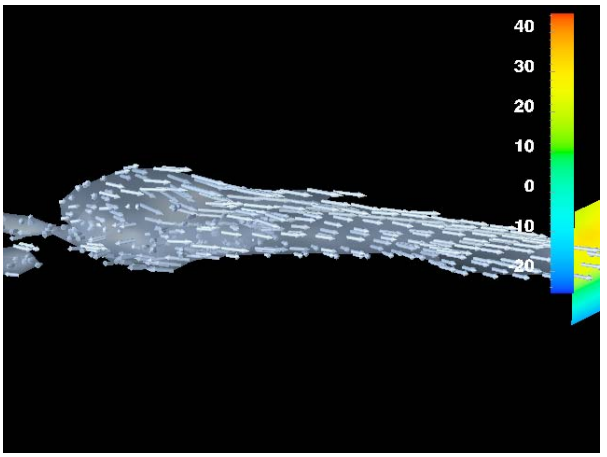


Figure 4.22
Final Image

Step 5.2 - Determine and understand the desired output's requirements for Vision 2

Suppose you want to save an animation file from the visual program in Figure 4.22 with resolution at 320 x 240 pixels. This requires you to determine a file name, set the resolution, and select an output format for the image.

- To save the visual program output, click on Save Image... in the File menu on the Image window. The save image dialog box will appear.
- Change the name of the *Output file name:* to "cloudwater.miff", change the *Format* to "MIFF", click on the Allow Rendering checkbox, change the *Image Size* to 320 x 240, click on the Continuous Saving checkbox, and click the Apply button (Figure 4.23). Pull up the Sequencer from the Execute menu and push the Stop button once (this will reset the Sequencer to have it start from frame 0.) Make sure the Palindrome button and Loop buttons are off; push the Play button; and wait for the sequence to finish.

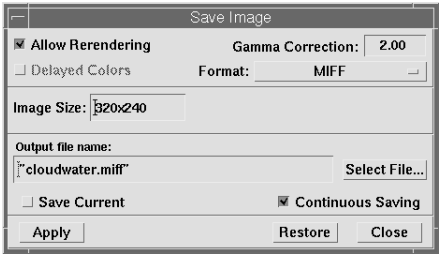


Figure 4.23
Save Image dialog box

Click the Continuous Saving button to turn it off and click the Close button to close the dialog box. The animation should now be stored in your directory as "cloudwater.miff". If ImageMagick* is installed, you can use it to view the animation and convert the sequence of images to a single MPEG "movie".

*ImageMagick is a public domain package for display and interactive manipulation of images. It is available at < http://www.imagemagick.org/>.

Second Hands-on Review

Within this lesson, you have done the following.

1. You used the **FileSelector** interactor to import data sets from different files.
2. You learned how to copy one or a collection of tools by using the drag-and-drop method.
3. You used the **Compute** module to compute simple expressions, including those that extract one of the elements of a vector or of the connection gridcounts. What else can **Compute** do? It has a large number of different functions, such as those that
   - cast between types (e.g. int(a), float(a));
   - do trigonometry and logs (e.g. sin(a), log(a), log10(a));
   - compute other mathematical functions (e.g. a+b, a*b, a^^b, exp(a), pow(a, b), sqrt(a));
   - perform vector functions (e.g. a dot b, a cross b, mag(a));
   - create random numbers (random(a, seed));
   - do vector construction and decomposition (e.g. [a, b], a.0);
   - compare using conditional functions (e.g. a ? b : c {interpreted as "if a then b, else c"});
   - compare strings (e.g. strcmp(a, b), strlen(a)); and more.

   Check the Compute manual page in the User's Reference for more details.
4. You created animations as sequences of images using the **Sequencer** to drive multiple image production and to supply different parameter values. In the example, the Sequencer output sets the "position" parameter of **Slab**, but it can be used in a variety of different ways. The sequencer just outputs an integer which can be used as
   - a variable to change- time (e.g. different series members);
   - viewpoint (e.g. different camera angles for a fly around effect);
   - variable (e.g. different isosurface values);
   - geometry (e.g. different slice planes); and more.
5. You learned to combine the **Sequencer** value with a **Compute** module to help you manipulate the integer output values into whatever form you want. For example if you need 10 values that vary from 0 to 1, you can use the expression "a/9.0" to set the range of the **Sequencer** between 0 and 9. If you need 10 logarithmic steps between 0 and 2, you can set the expression of the **Compute** to log(a+ 0.00001) and run the **Sequencer** between 0 and 90 in steps of 10.

6. You used **Inquire** to automatically determine how many grid counts (for possible "slab" points) are in a data set. The **Inquire** module can also be used to determine a variety of other information about an object, and can be helpful in determining relevant information necessary to set data-driven interactors or for conditional execution. The following are a few examples of information available.
   - How many members are in a group?
   - How many members are in a series?
   - How many items are in the 'data' component?
   - What are the dimensions of the 'positions' component (the grid extents)?
   - Is the 'data' component scalar? vector?
   - Is the 'data' component of type float? byte?
7. You used **Slab** to select a 2-D subset of a volume along a dimension.
8. You used **Colorbar** to display the colormap. Remember how you set the minimum parameter of **AutoColor** to make it use the same colormap for all the slabs. This allows you to use a consistent colormap for a number of images. On the other hand, sometimes a different colormap for each image provides higher contrast and gives a more "dramatic" effect or shows more information.
9. You saw that modules work fine when the data change from scalar to vector; for example, **AutoColor** operates on the magnitude of either scalar or vector data.
10. You used transmitters, receivers, and pages to organize the visual program. This is particularly important as networks get big, or if you are sharing networks with other people.

First Independent Exercises 5

Rationale

Now you have learned enough about OpenDX to change and extend visual programs on your own. For this chapter, you should first try to complete the two exercises without using the detailed step-by-step instructions. Then, you can work through the example solutions in detail, comparing your solutions with the provided alternatives. Topics covered in these exercises include: reloading a visual program, modifying captions, adding interactors, saving images, modifying compute expressions, coloring glyphs, and adding axes to an image.

Exercise 1. Extending the Sealevel Example

When presenting a visual analysis exaggerated scales can often help highlight data relationships. To illustrate, exaggerate the topological relief in the sealevel visualization. Note that after modifying the scale, the default positioning of captions is not ideal for the new depiction, so move the caption.

Open the visual program you saved in Chapter 3 ("sealevel.net"). Move the caption to the top left corner by modifying the *position* parameter. Use the online help if necessary. Change the size of the caption from 15 pixels to 25 pixels. Change the font to one of those listed behind the ... button at the far right side of the font parameter. Add another **Scalar** stand-in and pass its output to the *scale* of **RubberSheet**. Add this interactor to the existing control panel. Set the attributes on this interactor to go from a minimum of 0 to a maximum of .01 by an increment of .001. Choose Execute on Change from the Execute menu, and then experiment with changing the **RubberSheet** scale parameter using your new interactor. Save one of the resulting images in a TIFF formatted file.

Exercise 2. Extending the Cloudwater Example

In the cloudwater example, the slab was taken in the "x" dimension. This may not reveal the key features of the data set. Change the direction of the slab to highlight other information. Color plays a

unique role in visual analysis. It can catch the eye to important aspects of the data quickly. Change the network to color the wind glyphs for this purpose.

Open the visual program you saved in Chapter 4 ("cloudwater.net"). Put an axes box around the object. Change the *dimension* parameter of **Slab** to 1. Play the Sequencer and then reset its limits, noting that the limits of the Sequencer are incorrect now since there are fewer slabs in its new dimension than in the original example. For this data set, "dimension=1" does not imply the "y" dimension as you might expect. This is because the second dimension specifies the second-slowest varying dimension, which in this data set happens to be z (x varies slowest, z varies second slowest, and y varies fastest). Color the wind glyphs by passing the mapped isosurface through another **AutoColor** before passing it to **AutoGlyph**. Notice that you could also simply place the **AutoColor** after **AutoGlyph**–try this alternative (since **AutoGlyph** passes both the data and any colors through to its output, it doesn't matter which way you do it).

Step - by step instructions for Exercises

Instructions for Exercise 1

- If you need to start OpenDX, you can use the command "dx -edit" at the command line to start directly using the VPE. Use the File menu Open Program option. Select the program "sealevel.net" for which you saved in Chapter 3.
- Open the CDB for **Caption** by double clicking on the module. Change the *position* parameter to [.05 .95] to move it to the upper left-hand corner.
- Click the Expand button of the **Caption** CDB to view more Input parameters to caption. Change the *height* parameter from 15 to 25.
- Locate the *font* parameter, click on the "…" to the right of the current font name used and pick a new font such as "area" from the list (Figure 5.1).

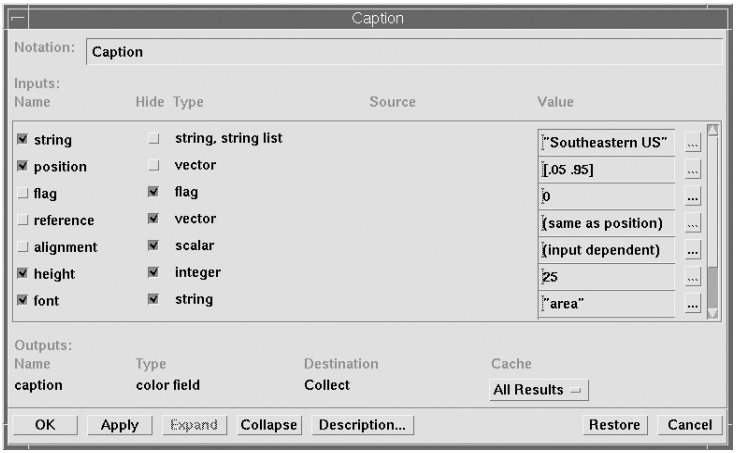


Figure 5.1
Final Caption CDB

- Click the OK button and execute the visual program (Figure 5.2).

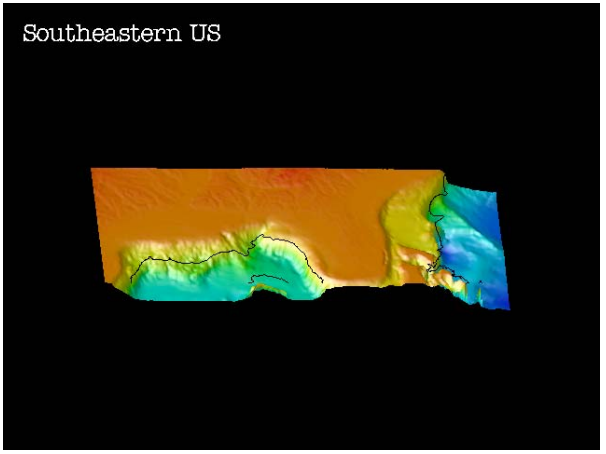


Figure 5.2
Final Image

- Place a **Scalar** interactor (from **Interactors**) just above the **RubberSheet**. Wire the output of the **Scalar** into the second input of the **RubberSheet**.
- Open the currently existing control panel by selecting Open Control Panel by Name -> Control Panel from the Windows menu. Use the middle mouse button and drag the newly added **Scalar** onto the Control Panel (Figure 5.3).
- Select the RubberSheet scale interactor in the Control Panel (click on its name). Select Set Attributes… from the Control Panel's Edit menu. Set the Maximum to 0.01, the Minimum to 0.0, and the Global Increment to 0.001. Make sure to press the Enter key after changing each value. Press the OK button.
- Select Execute on Change from the Execute menu. Use the **RubberSheet scale** interactor to change the value on the **RubberSheet**. Watch the image change with each click.

- After selecting one single image to output, select Save Image… from the File menu on the Image window. Click on the Save Current button, change the Format to TIFF and change the output name to "sealevel2.tiff".
- Click the Apply button. Once the cursor changes back to a pointer, the image has been saved and you may close OpenDX's windows. You can save the program at this time.

The scale between the data values in the data set and the grid size are typically not the same. Thus, when **Rubbersheet** is applied, a scaling factor is automatically determined that will make the 3-D object "look good". The scaling factor allows the programmer to increase or decrease this effect. It is possible to create a **Rubbersheet** function that would scale the data accurately given the grid and data value scales are known.

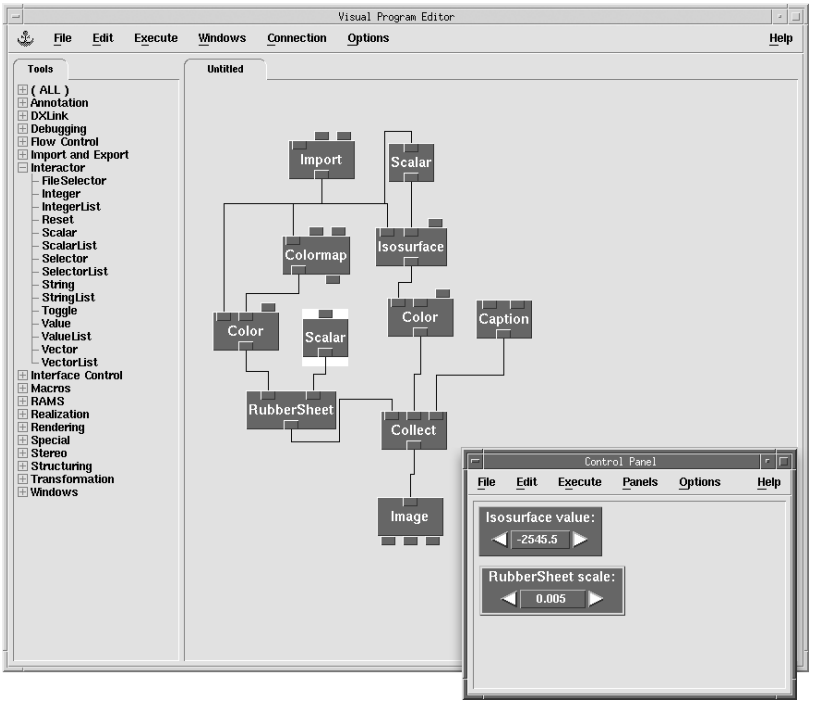


Figure 5.3
Final Visual Program

Instructions for Exercise 2

- If you need to start OpenDX, you can use the command "dx -edit" at the command line to start directly using the VPE. Use the File menu Open Program option. Select the program "cloudwater.net" for which you saved in Chapter 4.
- Execute the program once. Select AutoAxes… from the Options menu on the Image Window. Click on the AutoAxes enabled button. Click the OK button and execute the program once.

- Select the page with the **Slab** and **Colorbar** on it.
- Double click on the **Slab** to open its CDB. Change the *dimension* parameter to 1 (Figure 5.4).

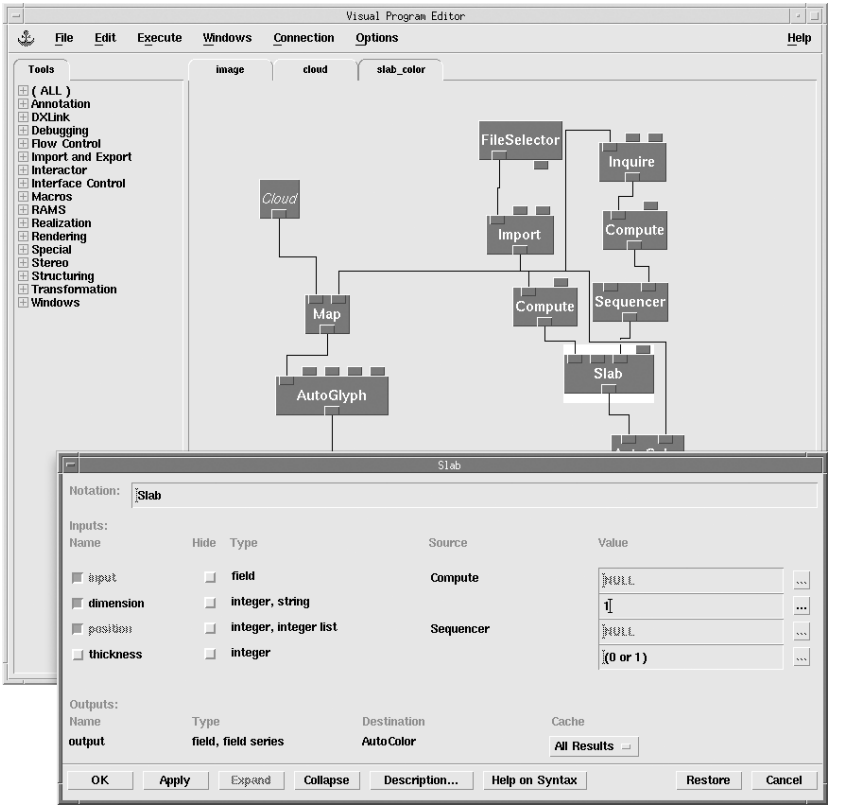


Figure 5.4
Slab CDB

- Click the OK button and execute the program with the **Sequencer**. An error message will occur because the number of slabs in this dimension is fewer than that in the original example (14 instead of 25).
- Double click on the **Compute** module located below the Inquire module to open its CDB. Change the *expression* to "a.1-1" so that the **Compute** selects the second dimension grid count value (Figure 5.5).

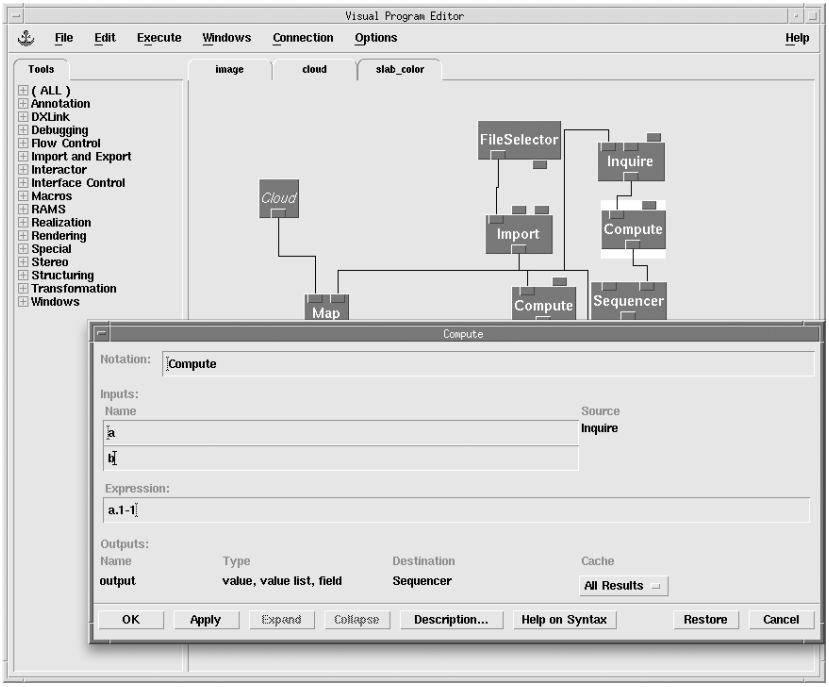


Figure 5.5
Compute CDB

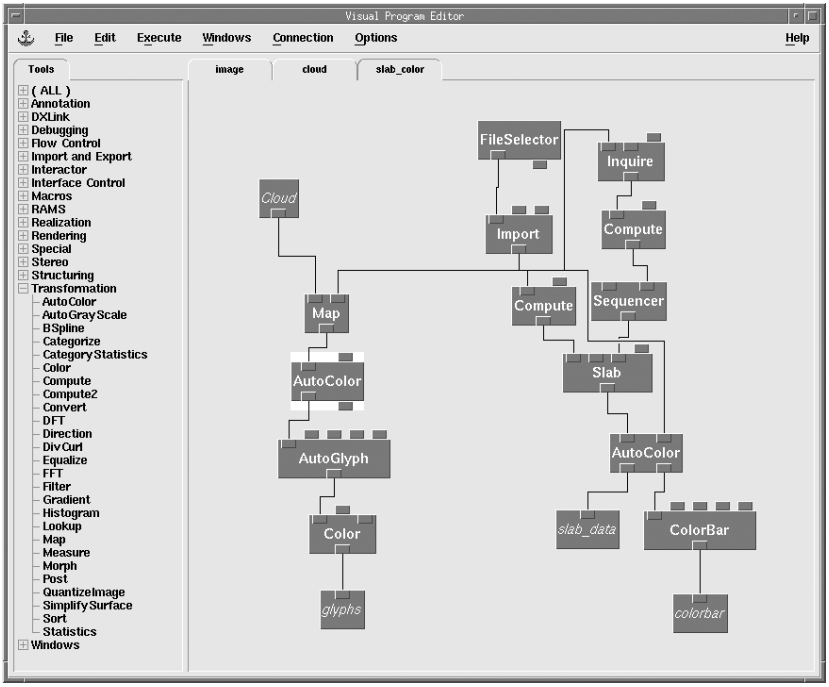
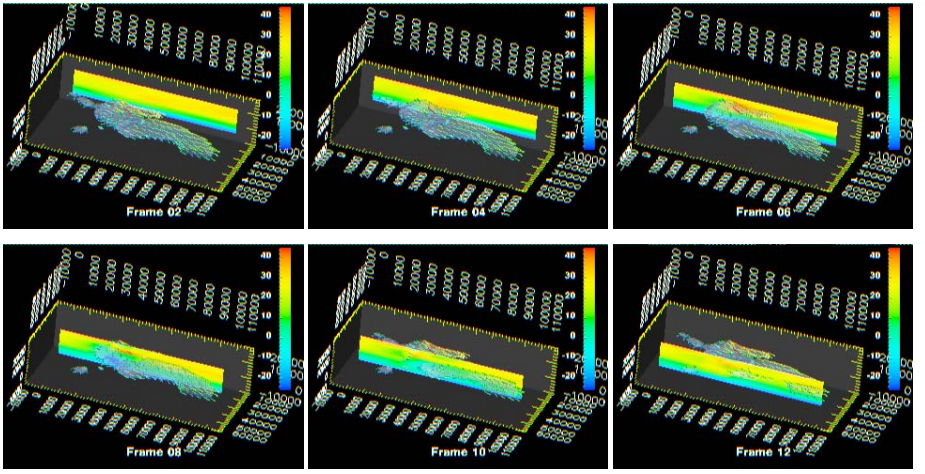


Figure 5.6
Final Visual Program

- Place an **AutoColor** module on the canvas between the **Map** and **AutoGlyph** (Figure 5.6).
- Execute the program with the **Sequencer** (Figure 5.7).
- You could also place the **AutoColor** after **AutoGlyph**–try this alternative, i.e., unhook the **AutoColor** from the **Map**, unhook the **AutoGlyph** from the **AutoColor**, and unhook the **Color** from the **AutoGlyph**. Rewire the network so that the order of the modules is **Map**, **AutoGlyph**, **AutoColor**, **Color**. Execute once.

Figure 5.7 *Final Animation*



Conclusion

Knowing how to determine the order for each module in a network is not clear at this point. Some modules should be starting to become familiar but creating your own program may still seem out of reach. Understanding the data model underneath all of the visual programming will help you to understand exactly how OpenDX works, which in turn will help you understand exactly how OpenDX modules transform data sets to go from their input form to the desired output form. Starting with the next chapter, you will begin to look at the computations that occur within OpenDX to help you develop a better understanding of the complete visualization process.

Mystery Data 6

Rationale

Visualization experts are often called upon to help other scientists to import and visualize their data. The scientists may not be able to completely (or even consistently) describe how their data set is organized, but the information may be available, embedded inside the data file. In this chapter, you are challenged to import a set of data files for which complete information is not available in advance, and without step-by-step tutorial instructions. You should start by working through both exercises. Then, you should work through the step-by-step instructions, comparing the details of your actions with the details of the solution suggested here.

Understanding the way that OpenDX organizes data is important to understanding how both the system and modules function. After importing the data, you should follow the program presented at the end of the chapter to learn more about how the data are organized in OpenDX.

Topics covered in these exercises include use of the Data Prompter and an introduction to the internal data structure of OpenDX.

Exercise 1. Mystery 2-D

In this exercise, you perform your first independent data import, then use the imported data to create a simple colored image. The name of the file that you are to work with is "mystery2d.txt". At the start, all that is known about this file is that it contains data on a regular grid and there is only one data value per grid element. However, other information about the file is assumed to be in the file itself. The goal is to use the Data Prompter to examine the data file, gather the information needed, collect it in a separate header file named "mystery2d.general", import the data, and produce an appropriate image. The visual program can be quite simple: import the data file, autocolor the data, then display the result as an image. If the data set is described and imported successfully, you should recognize the resulting image. Save your program as “logo.net”.

Exercise 2. Mystery 3-D

As in Exercise 1, you are given a data file with exact contents and format unknown. This time the file is named "mrb.binary". The only known information about this file is that it contains a three-dimensional data set aligned to a regular grid. However, other information is available as a header inside the binary file. Again, you must use the data prompter to create a header file that allows you to correctly import the data set into OpenDX. Name the header file "mrb.general". Check your import by pressing the "Test Import" button in the Data Prompter and compare the results to Figure 6.6.

Step - by step instructions for Exercises

Instructions for Exercise 1

- Start OpenDX using the Data Prompter. In UNIX, typing "dx -prompter" bypasses the initial startup dialog box, and starts OpenDX in a mode that gives you the Data Prompter.
- Select the file "mystery2d.txt" by using Select Data File... in the File menu. This file is included with the materials.
- The data are positioned on a regular grid; thus, select the Grid or Scattered file (General Array Format) option from the formats.
- To use the General Array Format option you must add information to the default settings, such as grid size, etc. To look in the data file and determine what other information is given, click on the Browse Data... button. This displays the contents of the file in a text viewer.
- Click the Describe Data... button.

Note: The data browser may disappear (automatically close). At any time, you may re-open it by selecting Browser… from the … button that succeeds the Data File (Figure 6.1).

- Look at the information from the header of the file. Enter those values appropriate to the file description into the dialog that creates a header file that describes this data, as shown in Figure 6.1. First, set the prompter to skip the header when it imports the data, i.e., because you are now using a separate, explicit header. Then, set the grid size as defined in the header, set the data format to ASCII, and adjust the data order. Save the explicit header file as “mystery2d.general”.

Figure 6.1 *Opening the Browser*

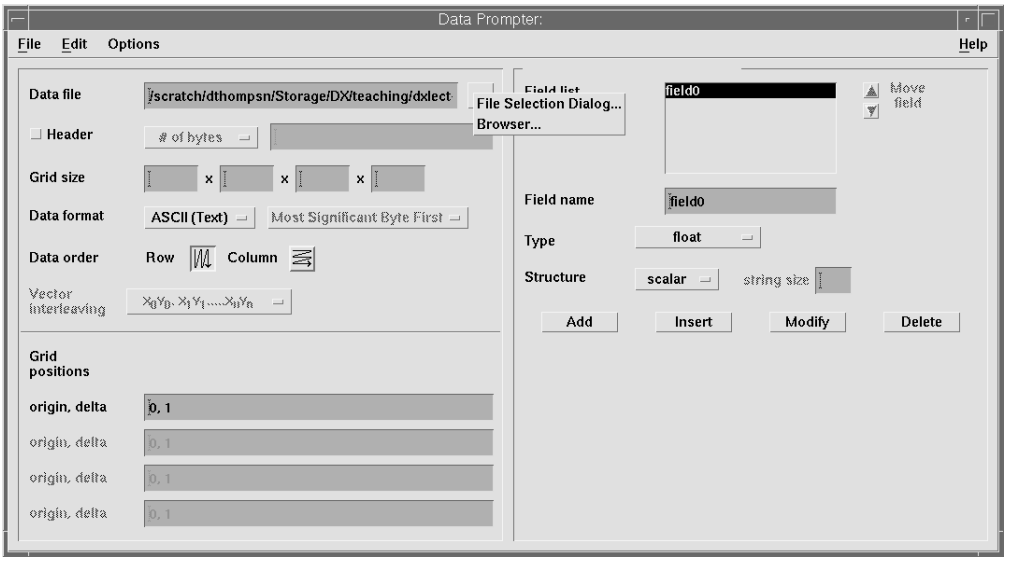
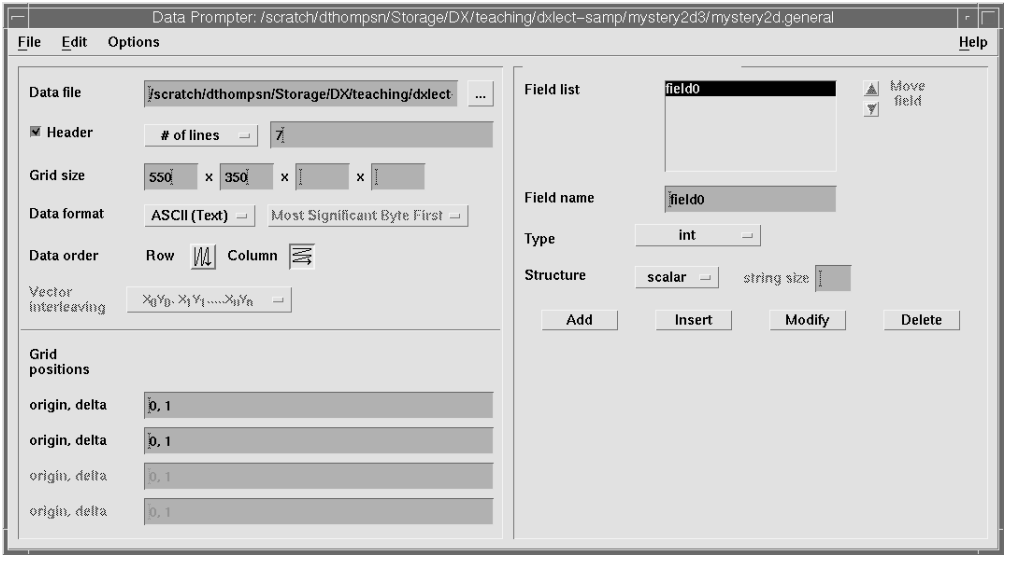


Figure 6.2 *Data Prompter Filled Out*



Start a new visual program in OpenDX. Do this easily at the UNIX prompt by typing "dx -edit" or double-click the icon in Windows.

- Place an **Import** module (from **Import and Export**) on the canvas. Set the file *name* input parameter to import the target data file.
- Connect the **Import** to an **AutoColor** module (from **Transformation**.)
- Connect the **AutoColor** to an **Image** module (from **Rendering**), to produce the program shown in Figure 6.3. Execute the program once.

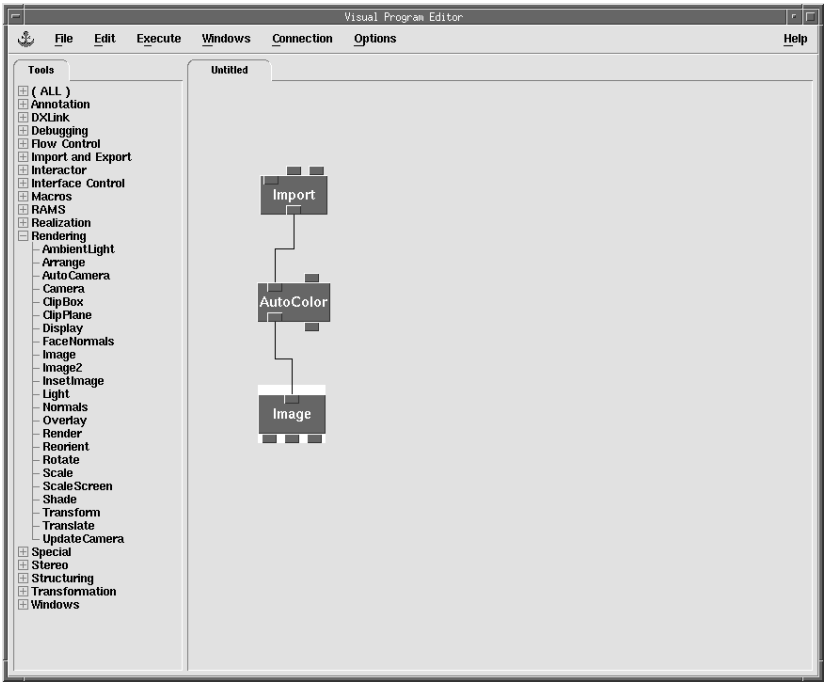


Figure 6.3
Complete Visual Program

Do you see a recognizable image (Figure 6.4)? If not, you may have to adjust your import header using the prompter.

- Save this program as "logo.net".

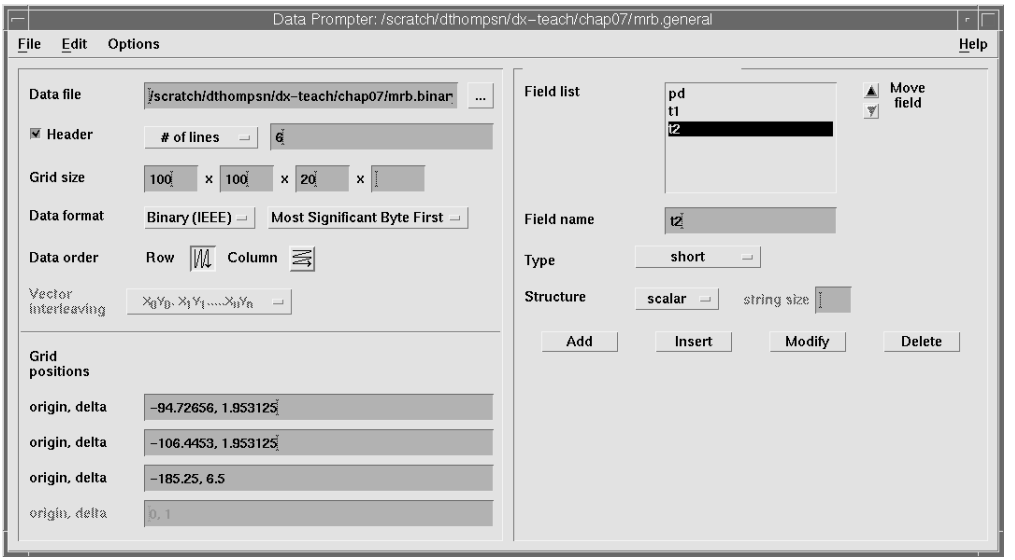
Figure 6.4
Resultant Image

Instructions for Exercise 2

- Start up OpenDX to use the Data Prompter, as in Exercise 1 ("dx -prompter"), or start OpenDX normally then choose Import Data from the initial dialog to start the Data Prompter.
- From the File menu of the Data Prompter, choose Select Data File… and choose the file named "mrb.binary" supplied with these materials.
- The file is known to contain data associated with a 3-D grid, so choose Grid or Scattered Data.
- Choose Browse Data... to look at the file. As you will see, the data in the file are encoded in binary, except for the first few lines. These lines represent an embedded header that gives the grid size and other characteristics of the data in the file.
- Press the Describe Data... button to get a dialog box that allows you to build an explicit header. Fill in the necessary information for the header. Make sure to set the data type to Binary (IEEE).
- Use the Modify and Add buttons to create the three short scalar variables.

Save the header using the Save As… command from the File menu, This data set is quite large, so it does not "auto-visualize" well. Use another function of the prompter called Test Import to assist in the import process, as illustrated in Figure 6.5.

Figure 6.5 *Completed Data Prompter*



Press the Test Import... button that is on the main Import Data window.

Figure 6.6 *Message window from Test Import*

```
Starting DX executive
Memory cache will use 112 MB (6 for small items, 106 for large)
port = 1900
 0:  worker here [18042]
server: accepted connection from client
Begin Execution
Object Description:
Input object is a Group which contains 3 members.
 member 0 is named pd
 member 1 is named t1
 member 2 is named t2
Each group member is a Field, the basic data carrying structure in DX.
The positions are enclosed within the box defined by the corner points:
[ -94.7266 -106.445 -185.25 ] and [ 98.6328 86.9141 -61.75 ]
Data range is:
minimum = 1, maximum = 2185, average = 356.979
(These are the scalar statistics which will be used by modules
which need scalar values.  The length is computed for vectors and
Begin Execution
the determinant for matricies.)
Input is not ready to be rendered because 3 Fields in the Input object do
not have colors yet.
Use the `AutoColor', `AutoGreyScale', or `Color' modules to add colors.
```

Compare the message window that appears to the information in Figure 6.6. If an error message occurs in the message window, you may have forgotten to set the data type to the right binary size.

Return to the data prompter and set the data type for all three variables to "short". You may have also forgotten to skip the 6 lines of the embedded header when you tried to import the file -- if so, correct that and try again.

Introduction to the Data Model

Understanding how data are structured and how OpenDX manipulates data internally provides you with significant insight into programming in OpenDX. The following visual program allows you to investigate how data are structured and to better understand how certain modules operate.

- Start a new session of the Data Explorer VPE by choosing New Visual Program from the startup dialog.
- Place an **Import** (from **Import and Export**) on the canvas, and fill in its *name* with the name of the file you saved from the prompter in Exercise 2. Change the *variable* parameter to "pd" (the first variable).
- Drag a **Print** module (from **Debugging**) onto the canvas and set its first input to the output of **Import**. Change its *options* parameter to "r", which means to recursively print the OpenDX object, and execute.

Take your time and thoroughly read the information presented in the resulting message window (Figure 6.7). Four components of the data set are described in detail: data, positions, connections, and box. This complete detail results from use of the recursive print option "r". The default for printing an object is "o", which just prints the top-level information about the object.

Figure 6.7 *Message window for "r" option*

```
Starting DX executive
Memory cache will use 112 MB (6 for small items, 106 for large)
port = 1900
server: accepted connection from client
 0:  worker here [15338]
Begin Execution
Field.  4 components.
Component number 0, name 'data':
 Generic Array.  200000 items, short, real, scalar
 Attribute.  Name 'dep':
  String.  "positions"
Component number 1, name 'positions':
 Product Array.  3 terms.
 Product term 0:   Regular Array.  100 items, float, real, 3-vector
 Product term 1:   Regular Array.  100 items, float, real, 3-vector
 Product term 2:   Regular Array.  20 items, float, real, 3-vector
 Attribute.  Name 'dep':
  String.  "positions"
Component number 2, name 'connections':
 Mesh Array.  3 terms.
 Mesh offset: 0, 0, 0
 Mesh term 0:   Path Array.  connects 100 items
 Mesh term 1:   Path Array.  connects 100 items
 Mesh term 2:   Path Array.  connects 20 items
 Attribute.  Name 'element type':
  String.  "cubes"
 Attribute.  Name 'dep':
  String.  "connections"
 Attribute.  Name 'ref':
  String.  "positions"
Component number 3, name 'box':
 Generic Array.  8 items, float, real, 3-vector
 Attribute.  Name 'der':
  String.  "positions"
Attribute.  Name 'name':
 String.  "pd"
```

Change the *options* parameter of **Print** to "rd", to recursively print the object description along with parts of the data. Clear the message window by selecting Clear from the Edit menu of the message window. Execute again. The result should look like Figure 6.8.

Figure 6.8 *Message window with "rd" option*

```
Begin Execution
Field.  4 components.
Component number 0, name 'data':
 Generic Array.  200000 items, short, real, scalar
 first 25 and last 25 data values only:
     14     13      7     11     19     20     10     12
     14     13     11     11      3      9     15     11
     12      9      9     10     19     11     14      5
     12
 ...                                                    15
      7      9     14     18      9     14     15     14
     10     11     14      9     17      9     11     18
      7      8     11     13     10     14     12     15
 Attribute.  Name 'dep':
  String.  "positions"
Component number 1, name 'positions':
 Product Array.  3 terms.
 Product term 0:   Regular Array.  100 items, float, real, 3-vector
  start value [ -94.7266, 0, 0 ], delta [ 1.95312, 0, 0 ], for 100
repetitions
 Product term 1:   Regular Array.  100 items, float, real, 3-vector
  start value [ 0, -106.445, 0 ], delta [ 0, 1.95312, 0 ], for 100
repetitions
 Product term 2:   Regular Array.  20 items, float, real, 3-vector
  start value [ 0, 0, -185.25 ], delta [ 0, 0, 6.5 ], for 20 repetitions
 Attribute.  Name 'dep':
  String.  "positions"
Component number 2, name 'connections':
 Mesh Array.  3 terms.
 Mesh offset: 0, 0, 0
 Mesh term 0:   Path Array.  connects 100 items
 Mesh term 1:   Path Array.  connects 100 items
 Mesh term 2:   Path Array.  connects 20 items
 Attribute.  Name 'element type':
  String.  "cubes"
 Attribute.  Name 'dep':
  String.  "connections"
 Attribute.  Name 'ref':
  String.  "positions"
Component number 3, name 'box':
 Generic Array.  8 items, float, real, 3-vector
 data values:
     -94.726562      -106.4453        -185.25
     -94.726562      -106.4453         -61.75
     -94.726562      86.914078        -185.25
     -94.726562      86.914078         -61.75
      98.632812      -106.4453        -185.25
      98.632812      -106.4453         -61.75
      98.632812      86.914078        -185.25
      98.632812      86.914078         -61.75
 Attribute.  Name 'der':
  String.  "positions"
Attribute.  Name 'name':
 String.  "pd"
```

Changing the print options to "rd" causes the object to be recursively traversed as before, but now a subset of the data for each component is shown along with the description. This allows you to

examine what is happening internally in the import process, without having to view every single data value.

- Add an **Isosurface** module (from **Realization**) between the **Import** and the **Print**. Clear the message window, execute once, then compare the result with Figure 6.8.

Notice that the data in the positions, connections, and box components are all different from the original values. Now the data component contains one unique value, and new components for colors and normals have been created.

What Isosurface has done to transform this three-dimensional data set, is find the point(s) where the data set has values that are equal to the value specified, which in this case is the data mean. OpenDX connects the points with equal values, thus giving new positions, connections and a box. Thus, Isosurface changes the data value to the specified data value and colors the defined surfaces with colors and normals.

- Connect the output of **Isosurface** to an **Image** module (from **Rendering**). Execute once.
- Place a **Remove** module (from **Structuring**) on the canvas and connect the **Isosurface** output to it. Place a second **Image** module on the canvas and hook the **Remove** to it. Change the *component* option of the **Remove** to "normals", Figure 6.9. Execute again.

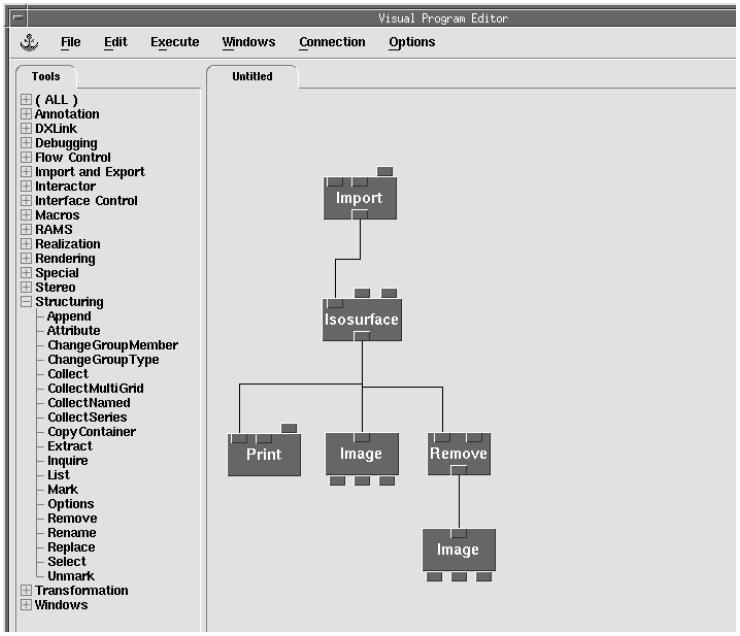


Figure 6.9
Current VPE

The separate Image modules create two separate displays (OpenDX allows you to define as many image modules as your machine's memory can handle). By comparing the two images, you should see

that removing the normals component deletes information that the renderer needs to shade surfaces. Without this information, all surfaces are rendered with the same color, resulting in a flat image.

Figure 6.10
Image with Normals and without

- Delete the connected **Remove** and **Image** modules.
- Add a **Color** module (from **Transformation**) below **Isosurface**. Set the *color* input to "red" and the *opacity* input to "0.5". **Send** the output of the **Color** module to a **Print** module with *options* "rd".

Look at the message window and notice how coloring has changed the Isosurface output, i.e., the 'colors' and 'opacities' components are changed (Figure 6.11). From this example you can conclude that the **Color** module affects only the two **components** 'colors' and 'opacities'.

Figure 6.11 *Message window showing changed components*

```
Begin Execution
Field.  7 components.

… (Components removed to reduce size) …

Component number 1, name 'colors':
 Constant Array.  42736 items, float, real, 3-vector
 constant value [ 1, 0, 0 ]
 Attribute.  Name 'dep':
  String.  "positions"

… (Components removed to reduce size) …

Component number 6, name 'opacities':
 Constant Array.  42736 items, float, real, scalar
 constant value 0.5
 Attribute.  Name 'dep':
  String.  "positions"
```

- Now place another **Import** tool on the canvas (the easiest way is by copying the one you already have; use the middle mouse button on it and drag to a new location). In the new **Import** module, import the "t1" variable.
- Place a **Map** module (from **Transformation**) below **Isosurface**; feed the output of **Isosurface** into the first input, and the output of the new **Import** into the second input. Move the **Print** to accept **Map**'s output and execute once.

Looking at the print after the **Map** (Figure 6.13), note that the only component **Map** changes is the 'data' component: a single constant value is replaced by a set of values dependent upon the isosurface positions. That is, **Map** looks at each position in the isosurface and finds what data value in the 't1' data set exists in the same space.

Place **AutoColor** below **Map** and print the output of **AutoColor**.

You know that **Color** only changes the 'colors' and 'opacities' components. **AutoColor** also only changes these two components. Looking at the 'colors' component, you will see that the color is no longer a constant value, but now has values that are dependent on the positions. Also note that the 'colors' component is stored as a floating point three-vector, which means OpenDX uses an internal 96-bit color scheme.

Wire the **AutoColor** into the **Image** to display the object that has been produced, Figure 6.12. Execute once.

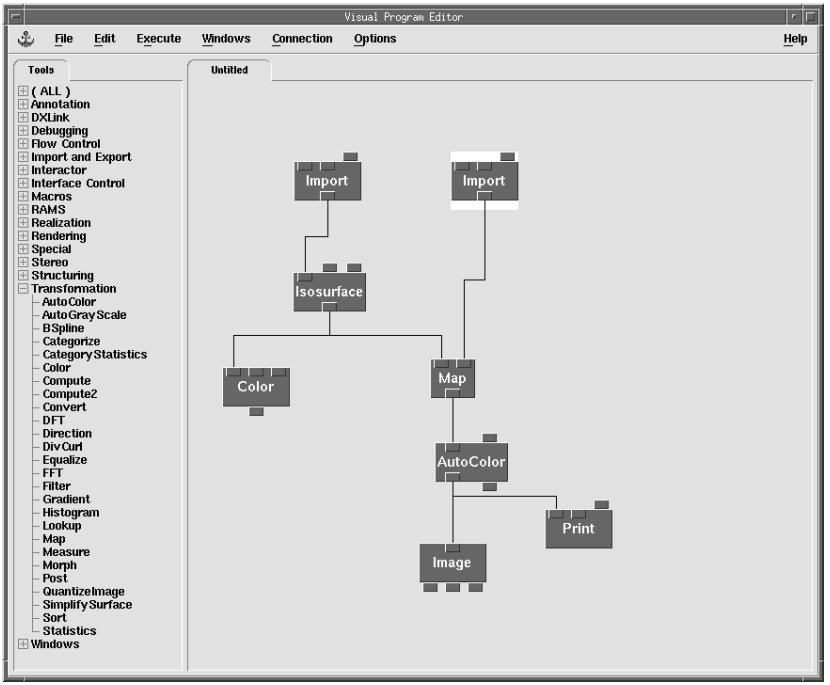


Figure 6.12
Current VPE

Put the output of **AutoColor** into the first input of a **VisualObject** (from **Debugging**). Run the output of the **VisualObject** into the **Image** module.

The **VisualObject** will produce a visual tree representation of the object's internal data structure. You may need to zoom in to view the information produced by this module.

Figure 6.13 *Message window showing mapped data*

```
Begin Execution
Field.  6 components.

… (Components removed to reduce size) …

Component number 2, name 'data':
 Generic Array.  42736 items, short, real, scalar
 first 25 and last 25 data values only:
    492    434    433    422    350    383    353    348
    406    349    329    428    323    315    377    322
    327    387    352    306    350    327    276    290
    300
 ...
                                                      521
    447    185    159    399    401    159    374    378
    359    340    318    312    345    351    372    357
    394    374    307    313    293    299    273    228

… (Components removed to reduce size) …

 Attribute.  Name 'der':
  String.  "positions"
Attribute.  Name 'Isosurface value':
 Generic Array.  1 item, float, real, scalar
 data values:
       263.6926
Attribute.  Name 'series position':
 Generic Array.  1 item, float, real, scalar
 data values:
       263.81586
```

Conclusion

All of the modules in the debugging category provide functions that can be used to illustrate how various modules transform an input data object to produce an output. The information provided by each of the modules vary; when in doubt, start with **Print** as one of the most general and useful debugging modules.

OpenDX has many strengths, but one of the biggest strengths is its very general and robust data model. Throughout this chapter, characteristics of the data model have been illustrated using the display produced by the **Print** module. In the next chapter, we describe the data model in much more detail, as a system of producing self-describing abstract objects. Understanding these details is a key in gaining further understanding into how parts of OpenDX perform their "magic".

OpenDX Data Model 7

Introduction

The OpenDX Data Model uses an object-oriented, self-describing approach to defining the data sets imported, used, and manipulated by the system. The data model is flexible enough to adapt to arbitrary or new types of data sets, generally using six types of descriptive objects.

An attribute names an association between an OpenDX object (array, component, field, or group) and a (simple or compound) value. A typical use for an attribute is to associate "metadata" with a data set.

An array object is a basic data carrying structure that holds actual data. OpenDX uses one-dimensional arrays and permits the array elements to be of any type, so an array object can be described by simply listing the number of items it contains. Array elements are referenced by index.

A component object is an element of a field with a specific role in data description; a component is typically an array object with a specific associated name.

A field object is a fundamental compound object in OpenDX, used to collect and encapsulate related components; all its elements must be components.

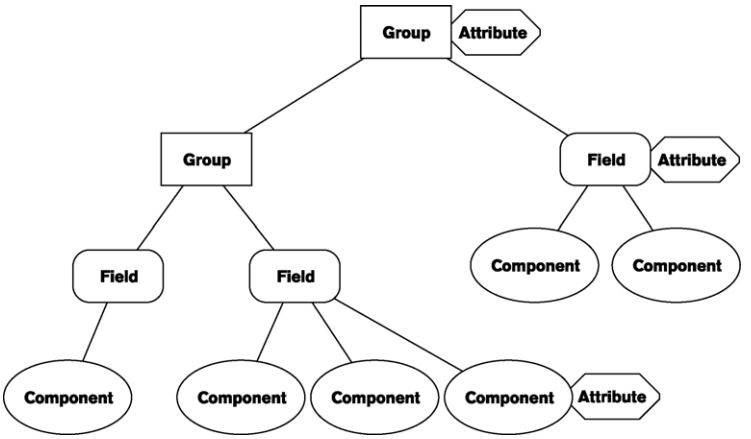
A group object is a compound object used to collect members that themselves may be fields and/or groups; it cannot collect components (a field is used for that purpose). A member of a group can be referred to either by name or by index.

OpenDX also uses other special purpose objects to describe special attributes or characteristics of objects used in the image rendering process, such as Camera, Light, Transform, and others. The most obvious sense in which the OpenDX data model is extensible is that new special purpose objects can be added to suit the purposes of modules with specific requirements. Any special object is simply ignored by modules that do not use or require it.

The diagram in Figure 7.1 illustrates how a somewhat complex data set might be described. As illustrated, fields collect components, groups collect fields and other groups, and attributes can be attached to arrays, components, fields, and groups. Remember that components are typically just named arrays that have a specific designated role in the data description, e.g., holding collections of

data values or coordinate system values, so the whole framework is just a way to systematically collect all the values and relationships present in a complex data set.

Figure 7.1
Object Diagram



Attributes

An attribute names an association between an object in the data model and a value that adds extra information (the value) to that object. The use of attributes formalizes the attachment of metadata to specific parts of a data set. OpenDX uses a set of standard predefined attributes and also allows users to define their own attributes. Predefined attributes are used frequently within OpenDX to define key characteristics of arrays, components, fields, and groups. For example, a subset of the predefined attributes used with components is listed below.

- 'dep' specifies the component on which the given component depends. For example, a 'data' component can be dependent upon 'positions'.
- 'ref' specifies the component to which the given component refers. For example, a 'connections' component will typically refer to the 'positions' component.
- 'der' 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 typically has a 'der' attribute naming the 'positions' component.
- 'element type' is an attribute of the 'connections' component. This attribute names the type of interpolation primitives.
- 'shade' indicates whether or not to shade the object if a 'normals' component is present.

Other predefined attributes, such as 'color multiplier', 'opacity multiplier', and 'fuzz' are described in the User's Guide under "Understanding the Data Model".

Array Objects

Array objects are the basic structures used in OpenDX to collect sets or sequences of actual information. An array consists of a designated number of items, referenced consecutively starting at 0. Generally the characteristics of an array object are given by the values of predefined attributes, such as type, category, rank, and shape. Each array definition is "self-describing", in the sense that both the array's data values and the values of its attributes are packaged in the same structure. OpenDX arrays can be heterogeneous, containing elements with different characteristics (vs. the homogeneous arrays that are required in most programming languages).

An example Array object description is "class array type float category real rank 1 shape 2 items 3 data follows 11 2.3 23 40 15 16". The standard OpenDX syntax has predefined attribute names followed immediately by the attribute's value. The type attribute of an array describes the internal numerical format to be used for the array's data. Predefined type values include *double, float, int, uint, short, ushort, byte, ubyte, and string*. The category attribute specifies which of two possible floating point representations is to be used, i.e., *real* or *complex*. The rank attribute refers to element order dimensionality, where rank 0 indicates a scalar, 1 a vector, 2 a matrix or rank-2 tensor, and 3 or higher a higher-order tensor. Lastly, the shape attribute defines the dimensionality in each of the order dimensions of the structure. Thus, 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.

Field Objects

A field object is a fundamental compound object in the OpenDX data model used to encapsulate a set of components (named arrays). Fields can also have attributes. Describing complex data sets that include both data elements and elements describing the space in which the data is positioned typically requires the specification of two or more arrays. Though you could use an array object to hold the data and then attach attributes (with compound values) to hold the spatial descriptions, OpenDX typically uses a field object to collect all this information in a set of specific, predefined component arrays. The following is a list of some of these standard field components.

- 'positions' stores the coordinates of a set of positions in an n-dimensional space. Except in special circumstances, this component is always defined.
- 'connections' provides a means for explicitly relating individual collections of positions (e.g., representing lines, surfaces, etc) and interpolating data values between positions.
- 'data' stores actual data values. Only one 'data' component can exist in a field, but other components can be used to store data.
- 'colors', 'front colors', and 'back colors' give specific information that helps the renderer determine how an object is to be depicted.

Other components, such as 'normals', 'opacities', 'box', 'faces', 'loops', and 'edges' can also be used to describe additional characteristics of complex data. See the User's Guide under "Understanding the Data Model" for a complete description of the set of OpenDX standard components.

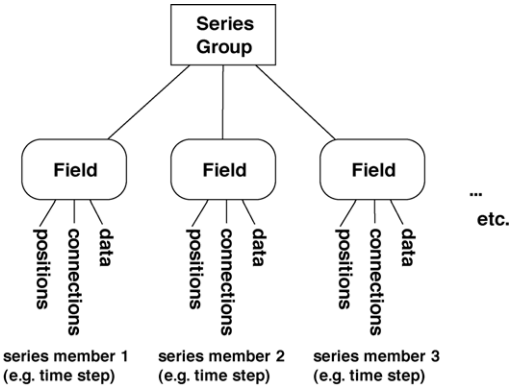
Group Objects

Groups are objects used to collect fields and other group objects together. Each object within a group is referred to as a member. Each member must have a unique numeric identifier, and may be given a unique name. When a group is used as an input to a module, the module normally examines the entire group object looking for appropriate components to manipulate. Thus, some group members may be modified while others are unchanged, but all are examined. It is acceptable to have a group with no members, though some modules will find such a group unacceptable.

There are four specific group types.

- A generic group is the standard group object used to collect related information.
- A multigrid group is a collection of separate fields, each with its own grid but treated as a single field, rather than as a group.
- A composite field group is a similar to a multigrid group, used primarily to segment fields to permit parts of the field/group to be processed in parallel in environments where parallel processing is supported.
- A series group is a generic group that stores a series value for each member in the group, as illustrated in Figure 7.2.

Figure 7.2
Simplified Series Diagram



Data Model Support

Intuitively, most data are collected for phenomena related to familiar types of geometric spaces. Thus, the OpenDX data model provides built-in support for a collection of commonly used geometric

structures. As noted above, this support is reflected in the various predefined attributes and components embedded within OpenDX that make it easy to define certain geometries. These include

- grid topologies such as lines, triangles, quads, cubes, tetrahedra, as well as faces, loops, and edges that are used for defining polyhedra; and
- position-dependent and cell-centered data support.

The data model also provides built in support for common varieties of data association, such as

- multi-dimensional and multi-parameter data sets; and
- data series (e.g. time dependent) data.

Generally the OpenDX data model is a self-describing object structure that allows the user to formulate flexible data descriptions that can be interpreted appropriately by general-purpose data transformation modules. The model also provides ways to detect and filter out invalid data (with special 'invalid positions', 'invalid connections', 'invalid faces', and 'invalid polylines' components) and to describe efficient methods to allocate and manage the memory required to store the described data.

How Modules Work

Most OpenDX Modules work in a similar manner: the module uses one or more group and/or field objects as input, manipulates certain components of the input objects to create one or more new objects as intermediate or final results, and makes some or all of its inputs and intermediate objects available as outputs. A module usually changes, adds, and/or deletes one or more components, but may also pass through other components unchanged. Each individual module uses and manipulates different components, as described in detail in its documentation. For example, the **Color** module only manipulates 'color' and 'opacity' components, and leaves other components ('data', 'position', 'connections', etc.) alone. Thus, the output of the **Color** module is a group/field object that is identical to the input object, except the color component has been changed or added.

Interoperability in OpenDX provided by the Data Model

The data model allows separate modules to work independently on different parts of an object. Each module looks for a specific component or components on which to operate. Since modules require and work on only certain components, the order in which the user places modules in the network (visual program) may not matter; however, in other cases the order does make a difference, and dramatically affects the way an object is depicted as an image.

For example, imagine a streamline that coils through a data set. The user wishes to make the streamline easier to see and also wants to be able to view the data values as colors along the line. The user decides to use three modules to perform this function: **Tube** to make the streamline more prominent in the image, **Map** to map specific data onto the streamline, and **AutoColor** to color the streamline based on the values of the mapped data. The user can arrange these modules in six different

orders, but, since the requirement is that the data values of the field be colored along the streamline, only three sequences are feasible: <**Map**, **AutoColor**, **Tube**>, <**Map**, **Tube**, **AutoColor**>, and <**Tube**, **Map**, **AutoColor**>. Knowing how OpenDX works on components allows you to understand the impact of each of these three orders.

To elaborate, the sequences <**Map**, **AutoColor**, **Tube**> (Figure 7.4) and <**Map**, **Tube**, **AutoColor**> produce identical output images, but <**Tube**, **Map**, **AutoColor**> (Figure 7.5) produces a somewhat different image result. To understand why, note first that the streamline prior to application of the **Tube**, **Map**, and **AutoColor** modules simply appears as a spiral or coil within a 3-D field (Figure 7.3).

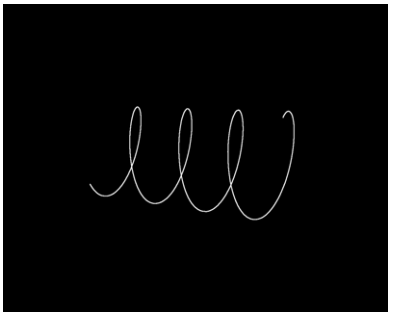


Figure 7.3
Original Streamline

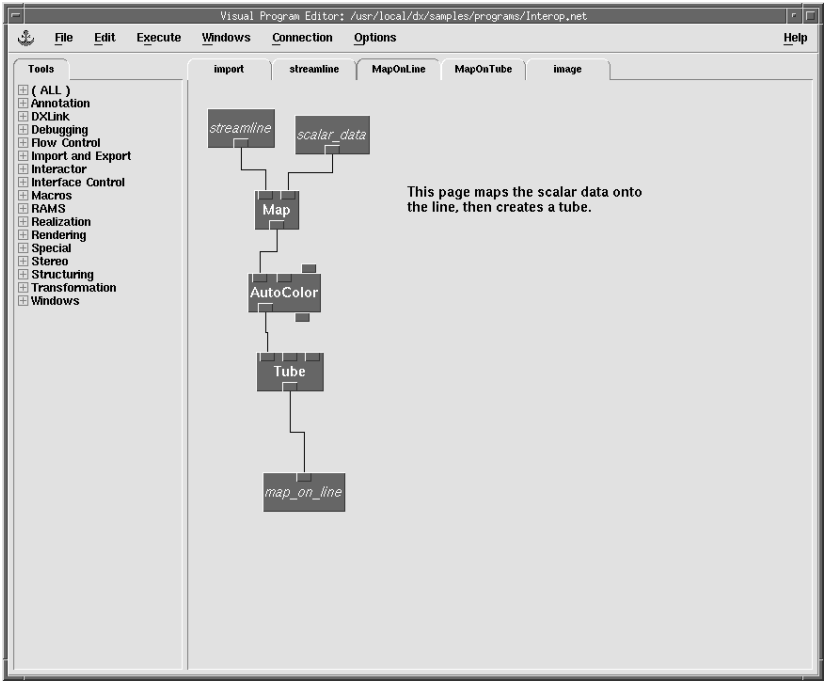


Figure 7.4
Map, AutoColor, Tube

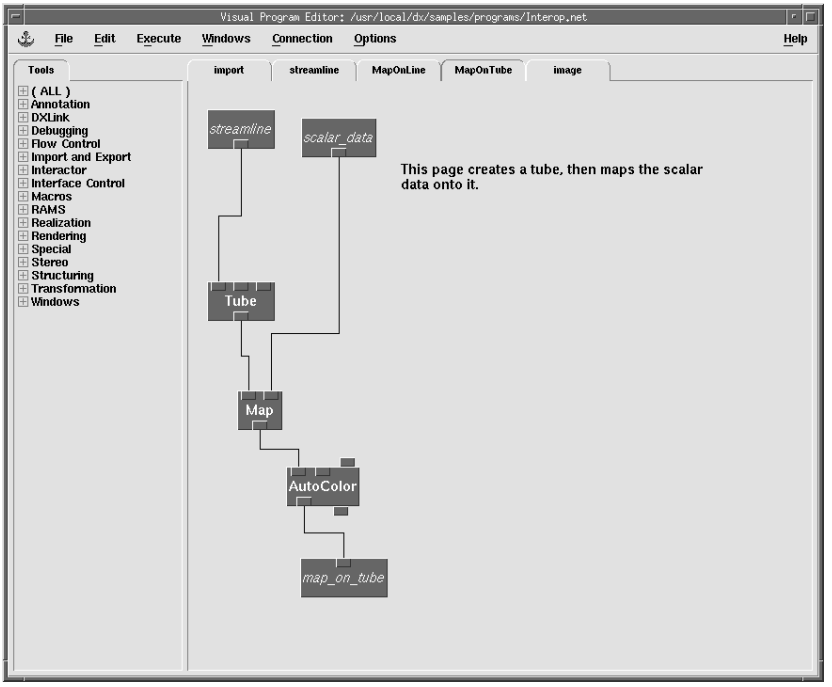
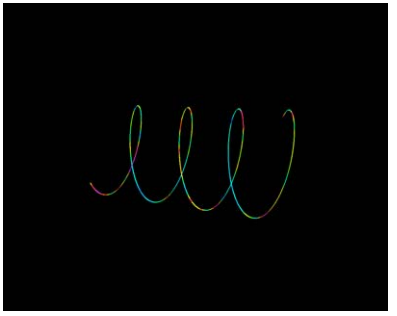


Figure 7.5
Tube, Map, AutoColor

Applying <**Map**, **AutoColor**, **Tube**> to the coil first maps the new data onto the streamline, then colors the coil to produce Figure 7.6. Now, tubing the coil (expanding the line's diameter) produces Figure 7.7, with colors in a concentric ring pattern.



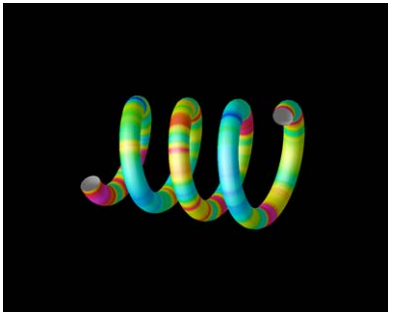


Figure 7.6
Colored Streamline

Figure 7.7
Tubed Result

Applying <**Tube**, **Map**, **AutoColor**> instead first tubes the streamline to produce Figure 7.8, with a default colored (thicker) coil. Now mapping the data to the coil then coloring the data produces Figure 7.9, with a very different coloring pattern on the coil.

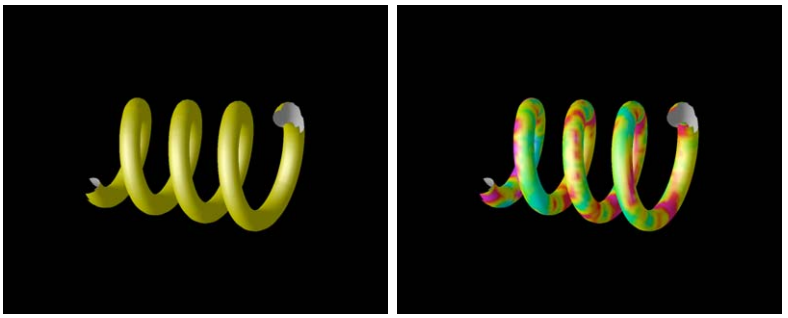


Figure 7.8
Tubed Streamline

Figure 7.9
AutoColored Result

The key to the difference between these two results is the role of the map operation. **Map** can match its data to a 2-D line or a 3-D surface. This flexibility makes it easy to use **Map** in a range of applications, but it is up to the user to determine the suitability of the application. In this example, the data to be mapped is such that it is most appropriately mapped to the 2-D line, giving the result shown in Figure 7.7; the approach resulting in Figure 7.9 would most likely be appropriate if the data to be mapped to the streamline were too complex to be represented on a simple line.

Summary

This is just a first step into understanding how OpenDX really works. With this basic introduction to the data model, you should begin to use and study the debugging information carefully, to identify exactly what data objects are present at each step and exactly how various modules work. Realizing exactly how each module changes, adds, and/or deletes data elements allows you to understand exactly how OpenDX works, and to start putting modules in the order required to produce a specific desired visual output. Reading the outputs in the message window from a **Print** may seem tedious, but it will help you gain a greater understanding of how OpenDX works with its data model. In the next chapter, you will look at the OpenDX data model of a two-dimensional grid and use the debugging information to change the data values and move the grid positions in space.

Manipulating Data 8

Rationale

Most visualization work requires that data be manipulated to facilitate the production of the desired visualization product. The previous chapter gave you an overview of the OpenDX data model. Next you will learn how to manipulate data within that model to achieve a desired result.

The first exercise illustrates how to create a network that manipulates 2-D grid positions to create a 3-D model that depicts a waving flag. The second exercise shows how to detect, handle, and manipulate invalid data.

Exercise 1. Flag Waving

In this exercise, you will examine the organization of the data model in more detail to see how the modules **Compute**, **Mark**, and **Unmark** work.

- Reload the program logo.net produced in Chapter 6.
- Hook the output of **Import** into the first input of a **Print**. Make sure that the *options* of **Print** are "rd". Execute once. Pay close attention to the 'data' component.
- Place a **Compute** module (from **Transformation**) on the canvas. Hook the output of **Import** into the input of the **Compute**. Change the *expression* in **Compute** to "a+20". Use **Print** to print the output of **Compute**.

Look specifically at the data component.

As you can see, the only component that changed was the data component. By giving the expression "a+20", each data point has its value increased by 20. That is, **Compute** applies its expression to every data value in a field, i.e., to every value in each 'data' component of an object.

Compute can be used to define a multitude of functions, using an expression syntax similar to that of the "C" programming language. Refer to the Reference Guide or the Context-Sensitive help for more details on writing expressions.

Many of Data Explorer's functions work primarily with the 'data' component, performing specific predefined calculations that create new components. For example, **AutoColor** creates a new 'color' component. Some modules, such as **Compute** and **Map**, actually change the content of the 'data' component. Other modules, such as **Isosurface** and **Rubbersheet**, change the 'positions' and 'connections' components.

The **Compute** module modifies data values, but it only works on the 'data' component. The next obvious question in data manipulation is how to change the 'positions' component with the **Compute** module.

- Replace the **Compute** module with a **Mark** module (**Structuring** category). Open the CDB of the **Mark** and change the *name* to "positions". Execute once.

The print in the message window shows that the 'data' and 'positions' are now identical, and the original data are now in a new component named 'saved data'. Thus, the **Mark** module copies the specified component into the 'data' component along with its original component name, and preserves the original data component as 'saved data'.

- Use a **Compute** module to change the values of this data. Place a **Compute** between the **Mark** and **Print** and enter the expression "[a.x, a.y, 30*sin(a.x/75.0 + a.y/100.0)]". Execute once.

Note the type of data that results. The *expression* in **Compute** creates a three-vector from the original two-vector. The values for the x and y positions are the same; however, they are explicitly written out, plus a new z component has been added. The a.x and a.y in the expression refer to the x and y components of the incoming vector data (this expression could have been written with a.0 and a.1 as well). This expression allows you to change the positions from 2-D to 3-D. However, now the positions are stored in the 'data' component, so you need to use the **Unmark** module to move them back to the 'positions' component.

- Add an **Unmark** module (from **Structuring**) and feed it the output from **Compute**. Add a **Shade** module (from **Rendering**) after the **Unmark**. Wire the rest of the program from the **Shade** into the existing **AutoColor** and **Image,** and execute once (Figure 8.1). Rotate the image a bit.

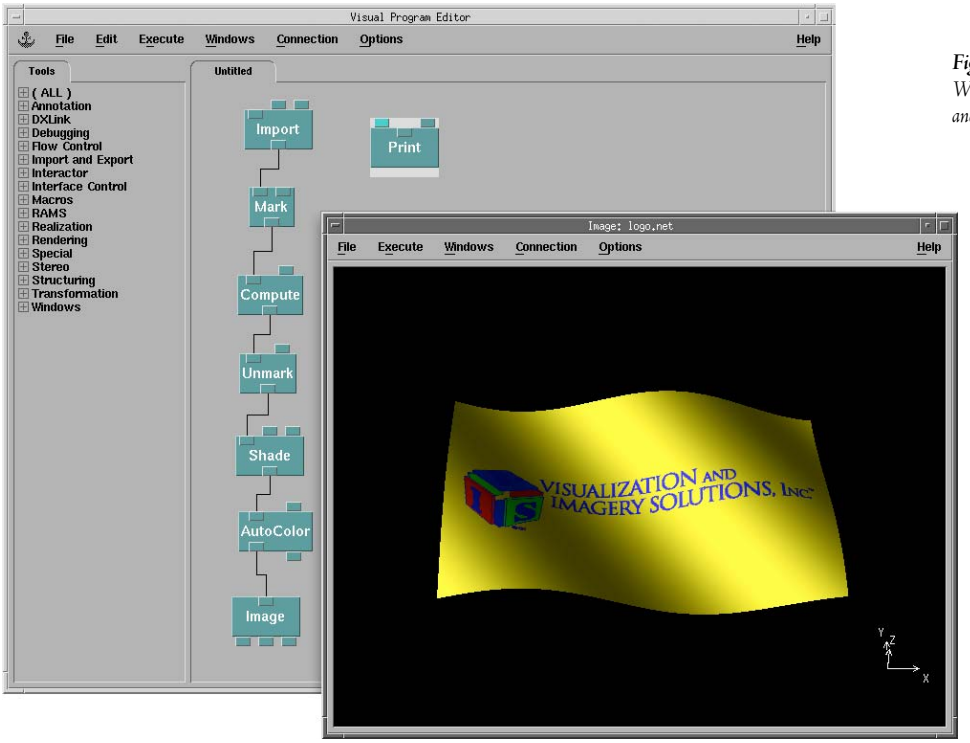


Figure 8.1
Warped Grid Image and Network

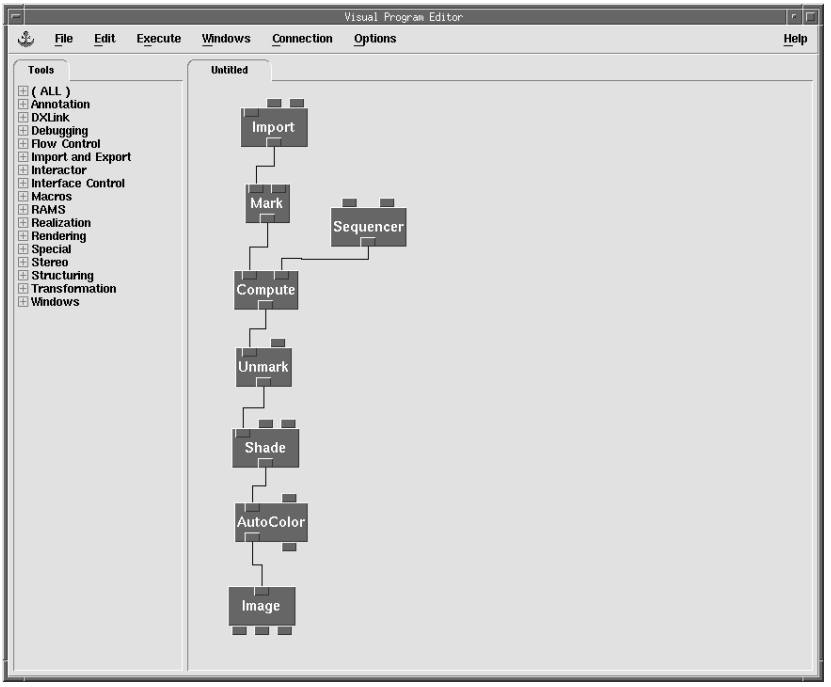
The **Unmark** module moves the 'data' component back to either the component originally marked, or to another named component, and restores the 'saved data' component to 'data'.

Originally, the positions were a regular grid, but now they have a "z" component that cannot be described regularly. This result is a new type of grid called a *Warped Grid*.

Note that you have to explicitly shade the object in this case, whereas the shading was added automatically in earlier examples. The reason is that in the earlier example the object was RubberSheeted; **Rubbersheet** automatically creates a 'normals' component, which adds the shading. Warping the positions does not create the 'normals' component, so you must explicitly use the **Shade** module to create the 'normals'.

Add a **Sequencer** (from **Special**) and run its output to the second input of the **Compute** (Figure 8.2). Change the limits on the sequencer to run between 1 and 10. Open the **Sequencer** by double clicking on the **Sequencer** stand-in. Click on the ellipses and the Frame Control will open. Change the *Max* to 10.

Figure 8.2
Final VPE for Waving Flag

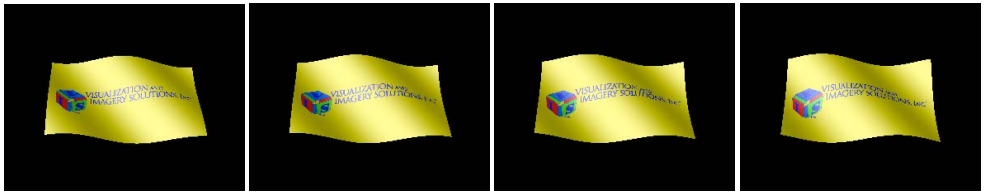


Change the *expression* of the **Compute** to "[a.x, a.y, 30*sin(a.x/75.0 + a.y/100.0 + b/10.0*6.28)]". Shrink the Image window to a relatively small size so that all ten images can be cached, then execute using the sequencer.

For the **Compute** module, each input is automatically labeled alphabetically from left to right, thus the first input is "a" the second input is "b", the third "c", and so on. So in this expression, the Sequencer is the variable "b". You can rename the inputs, but if you do, remember to use the new name in the expression. **Compute** is one of the modules that can have more input tabs added.

Once the images are each computed and cached, the flag with the VIS Inc. logo should wave in the wind.

Figure 8.3 *Waving Flag Images*



Save your program.

Exercise 2. Invalid Data

This exercise illustrates how OpenDX handles invalid data. Often model and field data sets sampled on a regular grid contain invalid or undefined data points. OpenDX incorporates a simple facility for dealing with this type of data.

"Bad", "missing" or otherwise invalid data are quite common in real data sets. Missing data values are sometimes marked by a special value, such as -9999. One method to handle invalid data is simply to remove all those values and proceed with the visualization. However, there are several reasons why it is better to mark these data points as invalid rather than removing them.

In a regular grid the positions and connections can be stored compactly (i.e. the positions can be specified by origin and delta, and the connections by the counts in each dimension). This encoding provides a tremendous saving in memory over explicitly listing each position and connection. However, if bad values are actually removed, the result is scattered data that lacks regularity and loses the associated memory savings. Also, there are cases where it is useful to know whether and where invalid data occur. For example, you may want to visualize your valid data, but also show the areas of invalid data.

Data Explorer supports the concept of 'invalid positions' and 'invalid connections'. A field typically has a 'data' component that is dependent on either the 'positions' or 'connections' component. To indicate invalid data, the field also has either an 'invalid positions' component or an 'invalid connections' component. Any data associated with an invalid position or connection is itself considered invalid (whether you have the position or connection flavor of invalidity depends on the dependency of the data).

All Data Explorer modules detect and understand invalidity. Typically they simply ignore the invalid data (pretend it isn't there); however, the "missing" positions or connections remain in the field if you want to operate on them explicitly.

Start this exercise by importing the Data Explorer formatted file "invalid_field.dx". Start by just Printing the output of **Import** with the "r" *option* (Figure 8.4).

Figure 8.4 *Message window of "invalid_field" print*

```
Field.  5 components.
Component number 0, name 'positions':
 Product Array.  2 terms.
 Product term 0:   Regular Array.  75 items, float, real, 2-vector
 Product term 1:   Regular Array.  100 items, float, real, 2-vector
 Attribute.  Name 'dep':
  String.  "positions"
Component number 1, name 'connections':
 Mesh Array.  2 terms.
 Mesh offset: 0, 0
 Mesh term 0:   Path Array.  connects 75 items
 Mesh term 1:   Path Array.  connects 100 items
 Attribute.  Name 'element type':
  String.  "quads"
 Attribute.  Name 'dep':
  String.  "connections"
 Attribute.  Name 'ref':
  String.  "positions"
Component number 2, name 'data':
 Generic Array.  7500 items, float, real, scalar
 Attribute.  Name 'dep':
  String.  "positions"
Component number 3, name 'invalid positions':
 Generic Array.  302 items, integer, real, scalar
 Attribute.  Name 'ref':
  String.  "positions"
Component number 4, name 'box':
 Generic Array.  4 items, float, real, 2-vector
 Attribute.  Name 'der':
  String.  "positions"
Attribute.  Name 'name':
 String.  "invalid field"
```

Printing the description of this data set shows that it is a standard 2-D field, with the exception that 302 'invalid positions' exist.

AutoColor the output of **Import** and wire it to an **Image**. Execute once.

As you can see, a large portion in the middle of the image is black, a good guess that the 'invalid positions' occur in this area.

Now insert the **Remove** module (**Structuring** category) between **Import** and **AutoColor**. Remove the "invalid positions" component.

Clearly all of the original positions of the grid are present and contain data, but they just are not being drawn by the renderer where the 'invalid positions' component exists.

Remove the **Remove** module. **Print** the output of **Import** with the *options* "rd" (Figure 8.5).

Pay particular attention to the 'invalid positions' component in Figure 8.5. Notice that this component contains integers which 'ref' the 'positions' component. What do you think this means? What are those integers?

Figure 8.5 *Message window with options "rd"*

```
Field.  5 components.
Component number 0, name 'positions':
 Product Array.  2 terms.
 Product term 0:   Regular Array.  75 items, float, real, 2-vector
  start value [ 0, 0 ], delta [ 0, 3.02703 ], for 75 repetitions
 Product term 1:   Regular Array.  100 items, float, real, 2-vector
  start value [ 0, 0 ], delta [ 3.0202, 0 ], for 100 repetitions
 Attribute.  Name 'dep':
  String.  "positions"

… (Components 'connections' and 'data' removed to reduce size) …

Component number 3, name 'invalid positions':
 Generic Array.  302 items, integer, real, scalar
 first 25 and last 25 data values only:
      2716      2720      2721      2730      2731      2739
      2740      2752      2753      2754      2755      2756
      2757      2758      2759      2760      2761      2764
      2768      2769      2776      2777      2778      2779
      2780
 ...
                4080      4081      4084      4164      4184
      4264      4284      4364      4384      4464      4484
      4564      4584      4615      4616      4617      4664
      4668      4669      4684      4715      4716      4717
      4768      4769
 Attribute.  Name 'ref':
  String.  "positions"

… (Component 'box' removed to reduce size) …

Attribute.  Name 'name':
 String.  "invalid field"
```

Insert a **ShowConnections** module (**Realization** category) between **AutoColor** and **Image**. Execute once.

The **ShowConnections** does what its name implies, i.e., it simply creates lines between connected points.

Insert a **Reduce** module (from **Import and Export**) between **Import** and **Print**/**AutoColor**. Execute once.

It should be obvious that there are fewer connections in this image. By studying the positions component in the message window (Figure 8.6), you can see that there are approximately half the number of positions in each direction as you might expect (38 as compared to 75 and 50 as compared to 100). By default the reduce *factor* is set to 2. If this were changed to 4, what would you expect to see? Note also that the **Reduce** module calculates new positions using interpolation; it does not just sub-sample existing positions, such as taking every second or fourth one.

Figure 8.6 *Message window of reduced object*

```
Field.  5 components.
Component number 0, name 'positions':
 Product Array.  2 terms.
 Product term 0:   Regular Array.  38 items, float, real, 2-vector
  start value [ 0, 0 ], delta [ 0, 6.05405 ], for 38 repetitions
 Product term 1:   Regular Array.  50 items, float, real, 2-vector
  start value [ 0, 0 ], delta [ 6.10204, 0 ], for 50 repetitions
 Attribute.  Name 'dep':
  String.  "positions"

… (Components 'connections' and 'data' removed to reduce size) …

Component number 3, name 'invalid positions':
 Generic Array.  265 items, integer, real, scalar
 first 25 and last 25 data values only:
       657       707       658       708       659       709
       660       710       661       711       664       714
       665       715       666       716       669       719
       670       720       675       725       676       726
       677
  ...
      1132      1141      1142      1181      1182      1191
      1192      1107      1157      1108      1158      1109
      1159      1133      1183      1134      1184      1135
      1185      1207      1208      1209      1233      1234
      1235
 Attribute.  Name 'ref':
  String.  "positions"
… (Component 'box' removed to reduce size) …
Attribute.  Name 'name':
 String.  "invalid field"
```

Now change the *factor* parameter of **Reduce** to 4. Execute again.

Pay close attention to the 'invalid positions' component. If you look at the "Attribute" at the end of the component, you will notice that this value changed from 'ref' to 'dep' (Figure 8.7). Do you remember what these attributes do?

These two types of invalidity, 'dep' and 'ref', can refer to either position or connection dependent data. 'dep' invalidity means the "invalid" component is simply a list of bytes that are either 0 or 1. 0 means that the associated position or connection is valid, whereas 1 means that the associated position or connection is invalid. The bytes are in a one-to-one correspondence (that's what dep means) with either the "positions" or the "connections". In contrast, 'ref' invalidity means the "invalid" component is a list of integers. These integers are references (indices) to the specific positions or connections that are considered to be invalid.

Figure 8.7 *Message window after reduced by 4*

```
Field.  5 components.
Component number 0, name 'positions':
 Product Array.  2 terms.
 Product term 0:   Regular Array.  19 items, float, real, 2-vector
  start value [ 0, 0 ], delta [ 0, 12.4444 ], for 19 repetitions
 Product term 1:   Regular Array.  25 items, float, real, 2-vector
  start value [ 0, 0 ], delta [ 12.4583, 0 ], for 25 repetitions
 Attribute.  Name 'dep':
  String.  "positions"

… (Components 'connections' and 'data' removed to reduce size) …

Component number 3, name 'invalid positions':
 Generic Array.  475 items, unsigned byte, real, scalar
 first 25 and last 25 data values only:
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00
 ...
                                  00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 Attribute.  Name 'dep':
  String.  "positions"

… (Components 'box' removed to reduce size) …

Attribute.  Name 'name':
 String.  "invalid field"
```

- Figure the minimum and maximum data values for this data set. (Hint: You will need to use the **Statistics** and **Echo** modules).

Echo prints into the message window just like the **Print** module. You may need to disable the **Print**, clear the message window, and then execute once. If you have set everything up correctly, you should see "ECHO: 0.601579 373.134."

- Insert an **Include** module (from **Import and Export**) after the **Reduce** module. Read the man page for **Include** (select Context-Sensitive Help from the Help menu and then click on **Include**) to figure out how to remove all values near the top of the data range (values above 300).
- Print the output of **Include** (Figure 8.8).

It is quite apparent that something has changed, since the output now fills up the message window. There is no longer an 'invalid positions' component present and there is a 'positions' component that is explicitly listed. Thus, the **Include** has physically removed all the invalid positions from the data set.

Figure 8.8 *Message window after include*

```
Field.  4 components.
Component number 0, name 'positions':
 Generic Array.  310 items, float, real, 2-vector
 first 5 and last 5 data values only:
              0               0
      12.458333               0
      24.916666               0
         37.375               0
      49.833332               0
...
          149.5       223.99998
      161.95833       223.99998
      174.41666       223.99998
        186.875       223.99998
      199.33333       223.99998

 Attribute.  Name 'dep':
  String.  "positions"
Component number 1, name 'connections':
 Generic Array.  244 items, integer, real, 4-vector
 first 5 and last 5 data values only:
         0         1        25        26
         1         2        26        27
         2         3        27        28
         3         4        28        29
         4         5        29        30
...
       286       287       304       305
       287       288       305       306
       288       289       306       307
       289       290       307       308
       290       291       308       309

 Attribute.  Name 'element type':
  String.  "quads"
 Attribute.  Name 'dep':
  String.  "connections"
 Attribute.  Name 'ref':
  String.  "positions"

… (Components 'data' and 'box' removed to reduce size) …

Attribute.  Name 'name':
 String.  "invalid field"
```

Change the *cull* parameter of **Include** to 0. Execute again and study the message window (Figure 8.9).

Notice that the image did not change, but the structure of the object listed in the message window did. The object now contains an 'invalid positions' component and the 'positions' are listed in a compact grid description. The *cull* parameter has a default value of "on" (1), which explicitly removes invalid positions and connections. Setting *cull* to "off" results in an output object that retains invalidity information.

Figure 8.9 *Message window with cull set to false*

```
Field.  5 components.
Component number 0, name 'positions':
 Product Array.  2 terms.
 Product term 0:   Regular Array.  19 items, float, real, 2-vector
  start value [ 0, 0 ], delta [ 0, 12.4444 ], for 19 repetitions
 Product term 1:   Regular Array.  25 items, float, real, 2-vector
  start value [ 0, 0 ], delta [ 12.4583, 0 ], for 25 repetitions
 Attribute.  Name 'dep':
  String.  "positions"

… (Components 'connections', 'data', and 'box' removed to reduce size) …

Component number 4, name 'invalid positions':
 Generic Array.  475 items, unsigned byte, real, scalar
 first 25 and last 25 data values only:
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00
 ...
                                 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 01 01 01 01 01 01 01 01
 Attribute.  Name 'dep':
  String.  "positions"
Attribute.  Name 'name':
 String.  "invalid field"
```

- Replace the **ShowConnections** with an **Isosurface** (Figure 8.10). It will help to specify a number of contour lines; set the *number* parameter of **Isosurface** to 10. Execute once.
- Save your program as "invalid.net".

As you can see from the image, **Isosurface** finds equal valued data points and connects them together. However, isolines are not rendered for invalid positions. This illustrates the comment made earlier: all modules in OpenDX are designed to work with objects that contain "invalid data".

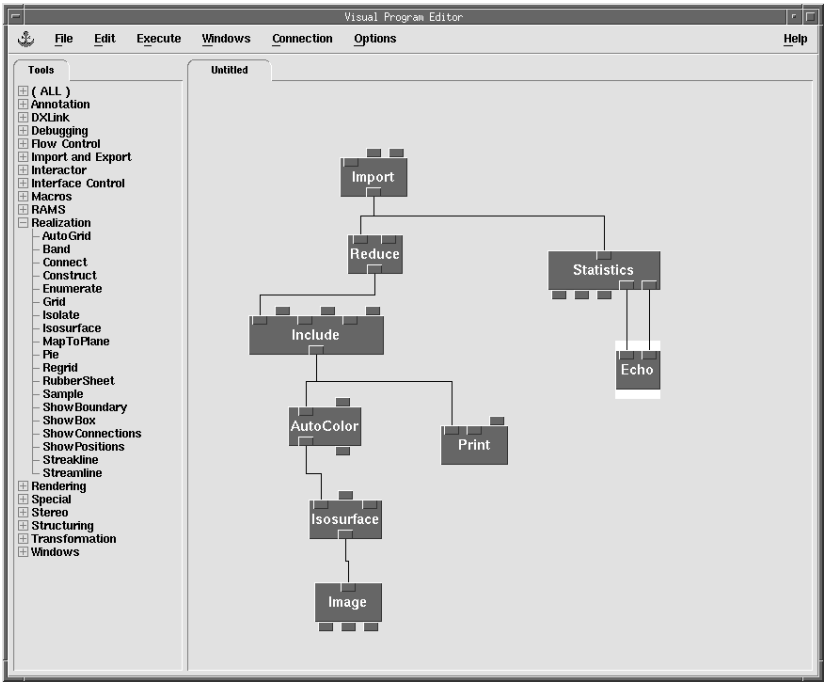


Figure 8.10
Final "Invalid" network

Conclusion

Often objects contain multiple data values stored as separate fields, such as those imported using the general array importer or two fields brought together with a **Collect** module. In this case, the user can use the **Select** module to select each variable. However, there are other times when it is more convenient to have multiple variables stored in the same field as separate components. To use the separate components in a field, you can use the **Mark** and **Unmark** modules as well as other modules not discussed in these exercises, such as **Extract**, **Rename** and **Replace**.

The **Compute** module applies its expression on every data value in the 'data' components. In order to compute on other components, **Mark** and **Unmark** modules can be used to temporarily interpret these components as 'data'.

With a combination of **Compute**, **Mark**, and **Unmark** you can construct your own **Rubbersheet** style module. There are times where you may wish to do this. With the combination of these three modules, you can construct a wide range of operations.

Compute is probably the most powerful function in OpenDX. Power users use it for everything from defining arbitrary mathematical transformations and string manipulations, to vector construction/deconstruction. You should familiarize yourself with its capabilities.

The **Remove** module does not work on data but works on the data model. Using **Remove** allows you to delete a specific component within a field. You used this module earlier to remove components such as 'invalid positions' and 'normals'. To remove data from a field, use the **Include** module.

If you import data with an "invalid" component, you can use either the reference or dependence method. Typically, however, you can create invalid data using the **Include** module. If your data has an invalid "marker" (such as -9999), you can use **Include** to mark the bad values as invalid. Which of the two types ("ref" or "dep") is used will depend on how many invalid items there are relative to the number of valid items. OpenDX will determine the optimum (memory savings) fashion to store the invalidity.

When working with large data sets that have a small number of invalid items, be sure to set the *cull* parameter of Include to 0 (the default is 1). As you have seen, the default behavior of **Include** is to cull (physically remove) invalid data values. This is generally not the best thing to do, because it results in non-regular data sets, which increases memory use. However, when removing a large portion of a data set using Include, the cull option may save some memory.

The strength of the data model is subtly shown in this chapter. Modules change only the components that they need to perform their function. For example, the **ShowConnections** module does not change the 'colors' component–the resulting image has the connections colored as before. The **ShowConnections** module creates a new 'connections' component and will only add a 'colors' component if one does not exist.

More on Data Import 9

Rationale

Often the use of scientific software is limited by the complexity of interchanging data from one format or software package to another. As new formats become *de facto* standards and new intermediate software packages become commonly used, visualization software is forced to adapt to different file formats. There are several different data organization strategies commonly used when storing information. Knowing the information's structure and being able to define that structure allows the researcher to describe data from a wide range of software environments, if the visualization package is flexible enough to accept general data specifications. OpenDX allows the visualization user to describe data formats in two different manners. The first is a simple, general array header file that describes predefined file structures. The second is the native file format facility that is flexible enough to permit definition of almost all data structures. We describe these two facilities in more detail below.

Data Organization

The way that data variables are organized in a file can make a difference in how that data set should be imported. The General Array Importer (Data Prompter) can import data that are arranged in two basic organization schemes, Block or Columnar (Figure 9.1).

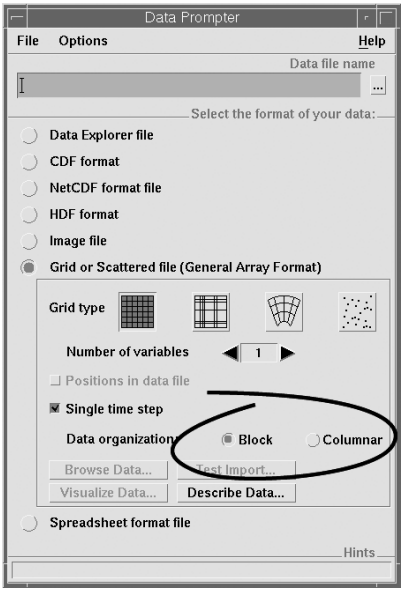


Figure 9.1
Data Prompter Organization Selector

Block style organization implies that the data are organized with variables in regularly sized blocks (or records) as illustrated in Figure 9.2.

Figure 9.2
Block organized variables

A1	A2	A3	A4	...
B1	B2	B3	B4	...
C1	C2	C3	C4	...

Columnar style organization implies that the data are organized with values in regular columns, like the spreadsheet output shown in Figure 9.3.

Figure 9.3
Columnar organized variables

A1	B1	C1
A2	B2	C2
A3	B3	C3
A4	B4	C4
...	...	...

An example of an actual data file arranged in the two different styles is shown in the following two figures. Note that each technique uses header lines before the actual block of data to describe the data organization.

Figure 9.4 *Example of a block style data set.*

```
grid_x = 5
grid_y = 4
origin_x = 2.1
origin_y = 3.0
step_x = 2
step_y = 1
100.9 99.9 102.3 106.8 109.9 99.1 100.2 100.1 102.1 98.1
99.9 109.3 103.1 108.1 99.9 109.1 90.2 112.3 107.2 108.2
7.7 8.0 9.2 1.2 2.0 1.2 3.0 1.8 9.3 9.2
1.2 3.4 8.3 1.3 5.4 1.9 9.2 9.1 1.0 3.1
```

Figure 9.5 *Example of a columnar style data set.*

```
grid_x = 5
grid_y = 4
origin_x = 2.1
origin_y = 3.0
step_x = 2
step_y = 1
temperature      pressure
100.9            7.7
99.9             8.0
102.3            9.2
106.8            1.2
109.9            2.0
99.1             1.2
100.2            3.0
100.1            1.8
102.1            9.3
98.1             9.2
99.9             1.2
109.3            3.4
103.1            8.3
108.1            1.3
99.9             5.4
109.1            1.9
90.2             9.2
112.3            9.1
107.2            1.0
108.2            3.1
```

Row versus Column Major Order

For multidimensional data with regular positions, you must specify both the data organization and the data order with respect to the position's axes. For example, two-dimensional data may be ordered such that the vertical (row) positions vary faster than the horizontal (column) positions, which is known as row major order. Alternatively, the horizontal positions may vary faster than the vertical positions, giving column major order.

To demonstrate row versus column major order, start with the regular grid illustrated in Figure 9.6, i.e., with origin at 0,0, deltas of 1 and 0.5, and counts of 5 x 4.

Figure 9.6
5 x 4 Grid

Assume that the data file lists data values as: 7, 12, 2, 17, 9, 10, 3, 11, 15, 4, 8, 14, 6, 1, 6, 13, 0, -1, 16, 5.

The "majority" keyword specifies how those data values are to be associated with the positions. If no explicit majority is defined, the default set by the Prompter is row major. Figure 9.7 and Figure 9.8 show how the list of values above would be associated with the 5x4 grid in row major and column major orders, respectively.

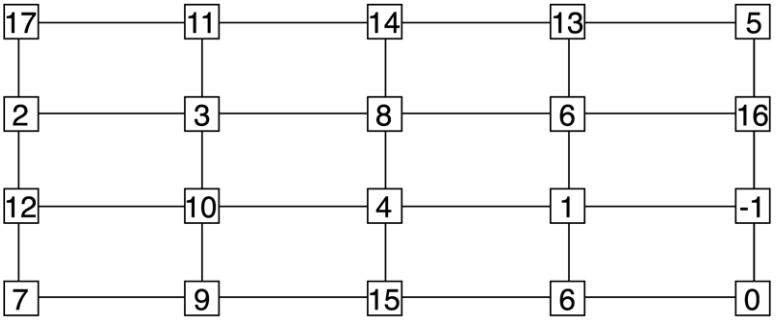


Figure 9.7
Row Major Order

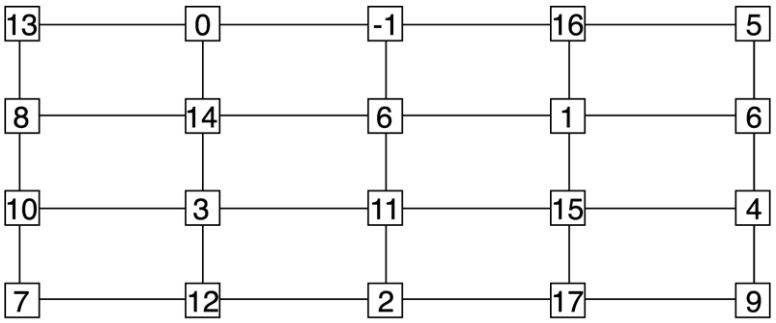


Figure 9.8
Column Major Order

Including Explicit Positions in a Data File

Data positions can be listed as explicit, primary values in an OpenDX data file, allowing for the specification of irregular, sparse, or so-called scattered positions. When scattered positions are included in a data file, the positions can be imported into Data Explorer using the General Array format and the Import module, or using the ImportSpreadsheet module.

If the General Array Importer is used, the keyword "locations" must be specified as the Field Name. Data Explorer then interprets values in the "locations" field as the positions of the data. The structure of the locations field would be 2-vector for 2-D data, 3-vector for 3-D data, etc. In Chapter 11, you will perform an import of a data set with the positions located in the data file.

If the data file is in a spreadsheet format, where each row contains the position and the corresponding data values, the ImportSpreadsheet module can be used to do the import. ImportSpreadsheet imports all columns as separate components of a field. Thus, the output of ImportSpreadsheet must be routed through the Mark, Compute, and Unmark modules to convert the column components into the "positions" component. The network needs one Mark module for each dimension of the data, with name parameters specified as "column1", "column2", and "column3" respectively for the successive Mark modules. Use Compute to form a vector [a,b,c] and Unmark to transfer the vector into the "positions" component.

General Header Files

Up to this point you have used the Data Prompter's graphical user interface to construct header files for the different data sets. These header files, usually suffixed by *general*, are ASCII text files that describe how the data are organized. After they are constructed and saved, these files can be modified as desired outside of the OpenDX system with any text editor. The data description language supported by OpenDX uses the same generic version of a header to describe multiple files, e.g., a descriptor with the same structural entries but with different sizes and value entries. This generic type of *general* header file is called a Template. If a *general* header file can describe the location of grid

information within your own data file, then OpenDX can use this information without needing to explicitly add it.

Templates

Typically data analysis, acquisition, and simulation programs are used to create a number of different data files, each with the same internal structure. Once you successfully describe a particular data format with the "Data Prompter", the filename may be the only variable that must change to allow a visual program to import one file or another. Templates allow you to construct one general header file that can be used for several data files.

Inside the **Import** module, the user fills in the information as follows:

name = data file name (not the *general* file)

format = "general, template = file-header.general"

If you use a template in this fashion and the header file contains the keyword "file=", OpenDX generates a warning to let you know that OpenDX is ignoring the filename. If you wish to remove the warning, you must delete the "file" entry from the corresponding *general* file.

If only the number of points or grid size changes from one data set to another, you can use a template to override these other parts of the header file as well. For example, if the grid in a data file is different from the template, then you can use the grid keyword in the Import Format input.

name = data file name

format = "general, template = file-header.general, grid = 13x14"

Deriving Grid Information

Often information about the grid, such as the number of points or the origin/deltas, is included as part of the data file itself. As illustrated in Figure 9.9 and
Figure 9.10, you can derive information about the grid, points, and positions by using the OpenDX keyword "marker" to associate an OpenDX keyword with an actual label that appears in the data file.

Figure 9.9 *Example Data File*

```
grid_x = 5
grid_y = 4
origin_x = 2.1
origin_y = 3.0
step_x = 2
step_y = 1
100.9 99.9 102.3 106.8 109.9 ...
7.7 8.0 9.2 1.2 2.0 1.2 ...
```

Figure 9.10 *OpenDX Header File with Derived Grid Information*

```
file = datafilename
grid = marker "grid_x = " x marker "grid_y ="
field = temperature, pressure
header = lines 6
positions = marker "origin_x =", marker "step_x =", marker "origin_y =",
marker "step_y ="
```

The Native File Format

OpenDX uses its own format to store data internally. This format can represent virtually any object that can be displayed or manipulated. By writing explicit external processing code, you can create data files in this format that can be directly imported into OpenDX. The native format can be used to describe some data types that cannot be imported using the General Array Importer, such as:

1. skewed grids;
2. combination grids where positions may be irregular in some dimensions but regular in others (often called "product arrays");
3. irregular connections (e.g., tetrahedra or triangles);
4. fields with components other than "positions", "connections", and "data";
5. data with required meta-data attributes;
6. compound structures known as faces, loops, and edges; and
7. specialty objects such as lights and cameras.

The **Import** module reads data in native OpenDX format directly, from either a file or as a stream from stdin. To execute a program that writes to the native OpenDX format on stdin, you include the bang "!" operator preceding the name of an external executable that creates the native format data stream. For example, you can use the set of data converters called "gis2dx" developed by the University of Montana Computer Science Department to produce a native format data stream by giving the **Import** module a Name parameter of "!gis2dx".

The OpenDX file format is similar in fashion to the internal storage of OpenDX objects. Each object is defined by a header section accompanied by an optional data section. The header section describes

the structure of the object, and the data section is simply an array object that defines the values in the object.

The OpenDX native file format can also refer to separate data files. As an example, consider a data set that lies on a regular grid, has triangular connections, and has data values at each position. The grid has origin {0, 0}, delta's {1, 1}, and has counts {2, 3}. The triangle connections for the grid are {1, 0, 3}, {1, 3, 4}, {4, 2, 1}, {5, 2, 4}. Remember OpenDX uses row major order by default, so the grid is organized as shown in Figure 9.11. The data values for the positions are {0.21, 0.07, 0.62, 0.88, 0.81, 0.005}.

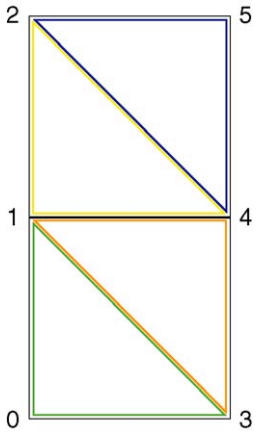


Figure 9.11
OpenDX Order of Points on Grid

To begin describing this data set as a native file, you first build the header section for the grid, as illustrated in Figure 9.12.

Figure 9.12 *Grid Header Section*

```
# Lines beginning with a pound sign are comments.
# Define the header for the positions.
object 1 class gridpositions counts 2 3
origin 0 0
delta 1 0
delta 0 1
attribute "dep" string "positions"
#
```

To complete the native file description of the unconnected set of positions you need to add the extra three lines shown in Figure 9.13 to add the formal definition of the object.

Figure 9.13 *Native file describing an object of 6 unconnected points equally spaced.*

```
# Lines beginning with a pound sign are comments.
# Define the header for the positions.
object 1 class gridpositions counts 2 3
origin 0 0
delta 1 0
delta 0 1
attribute "dep" string "positions"
#
object "default" class field
component "positions" value 1
end
```

Next you add the information defining the "connections" and the "data". For simplicity, you assume that the data values are contained in a separate file named *data.txt*. Figure 9.14 shows how to define the header with the connection data immediately following the grid points.

Figure 9.14 *Adding the connections to the native file.*

```
# Lines beginning with a pound sign are comments.
# Define the header for the positions.
object 1 class gridpositions counts 2 3
origin 0 0
delta 1 0
delta 0 1
attribute "dep" string "positions"
#
# Define the header for the connections
object 2 class array type int rank 1 shape 3 items 4 data follows
# List the data
1 0 3
1 3 4
4 2 1
5 2 4
# Add the necessary attributes
attribute "element type" string "triangles"
attribute "dep" string "connections"
attribute "ref" string "positions"
#
object "default" class field
component "positions" value 1
component "connections" value 2
end
```

The last piece of information to add is the data. As mentioned earlier, the data set is assumed to be in the file *data.txt*, thus the actual data description includes structural information, the term “data file”, and the actual file (Figure 9.15).

Thus, the user who is building external programs can create output data streams for OpenDX in the appropriate native file format. Or, separate programs such as “gis2dx” can be built to convert a data set in one format to a data set encoded in the OpenDX native file format. Thus OpenDX can accommodate most sophisticated data structures.

Figure 9.15 *Native file with Data added*

```
# Lines beginning with a pound sign are comments.
# Define the header for the positions.
object 1 class gridpositions counts 2 3
origin 0 0
delta 1 0
delta 0 1
attribute "dep" string "positions"
#
# Define the header for the connections
object 2 class array type int rank 1 shape 3 items 4 data follows
# List the data
1 0 3
1 3 4
4 2 1
5 2 4
# Add the necessary attributes
attribute "element type" string "triangles"
attribute "dep" string "connections"
attribute "ref" string "positions"
#
# Define the header for the data
object 3 class array type float rank 0 items 6
data file data.txt
attribute "dep" string "positions"
#
object "default" class field
component "positions" value 1
component "connections" value 2
component "data" value 3
end
```

Summary

The general array importer is flexible enough to accommodate a large number of data formats. It is possible to list the general header information directly above the data in a data file. So if you write your own file output routine in a program and do not need the power of the native file format, you can write the file directly to the general format.

With the flexibility of the template function of the general format, it is easy to bring in multiple data files with one single general description file.

The native file format is extremely powerful; however, it does require effort to understand its layout. For more information on how to construct Data Explorer formatted files, make sure to work through Chapter 15 and see the Appendices in the User's Guide. It is also possible to export OpenDX files as ASCII so you may model your own files after an existing OpenDX objects. To do this, use the Export module and change its format parameter to "dx text follows" or "dx text".

Network Flow Control 10

Rationale

OpenDX provides a visual programming environment that combines elements of data flow with more traditional conditional flow control. OpenDX's flow control modules allow a program to dynamically decide which branch of a visual program to execute, or to implement looping, routing, and switching operations. These operations are essential in building complex visual applications.

Since OpenDX is a complete visualization environment, there are a large number of modules available to perform visualization tasks. Knowing what each of the modules does is beneficial, but not necessary. Throughout the next few chapters, you will look at a few of the most common visualization tasks and the modules most often used to construct the visualizations within OpenDX. Not all modules are covered here, but additional details and examples are available with the OpenDX distribution.

The data set used in this chapter is courtesy of The Department of Radiology, New York University School of Medcine, 550 First Avenue, New York, NY 10016. Image format conversion was performed using Interformat, available from www.radio-logic.com.

Exercise

This exercise is designed to import and manipulate the three-dimensional MRI data set from Chapter 6 in various ways. First, import the mrb.binary data set into OpenDX and use the "pd" variable to create an isosurface of the exterior of the volume for reference. Second, make it possible to view arbitrary *slices* of a user selected data variable from the data set. Third, add a method to clip off portions of the data to allow better viewing. Fourth, create a second Image window that can be popped-up on demand using an interactor, to show the object rotated slightly from the original. Finally, use OpenDX's execution styles to investigate how a completed network runs as a developed program. You may also wish to attempt this exercise on your own. For this reason, we start by showing the completed Control Panel and final images in Figure 10.1 and Figure 10.2, respectively.

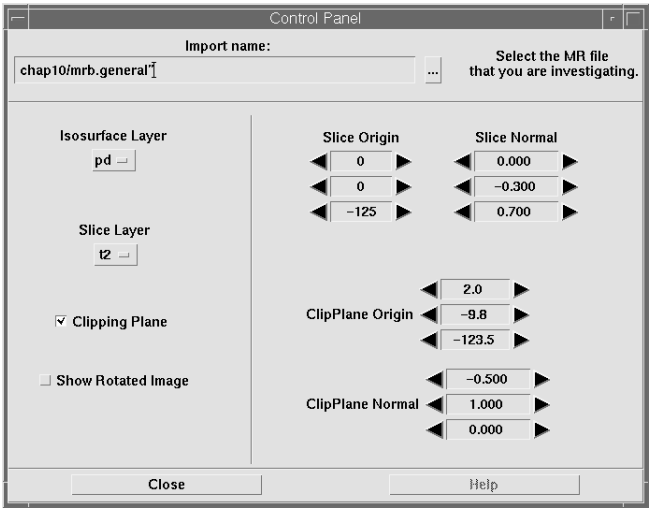


Figure 10.1
Final Control Panel for this exercise

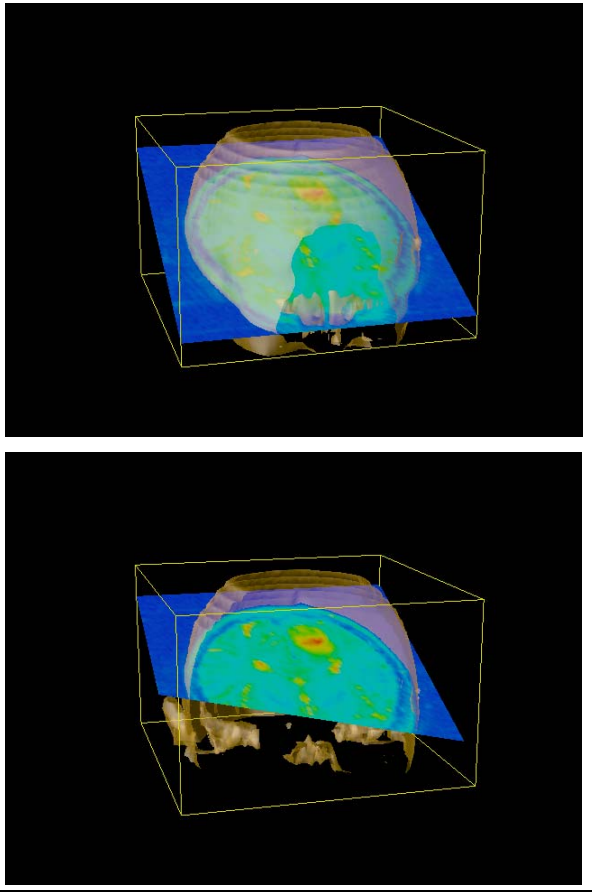


Figure 10.2
Final Images for this exercise

Step by Step Instructions

- Use a **FileSelector** interactor and an **Import** module to import the "mrb" file.

Recall that this file has already been imported in an exercise in Chapter 6, so simply import the header file created during that exercise. Remember this data file contains an object with three variables "pd", "t1", and "t2", and it is imported as an OpenDX group with three fields of data.

- Place a **Select** module (from **Structuring**) below **Import** and connect the output of **Import** to the **Select**.
- As an easy way to choose which of the three fields to depict, pass the output of **Import** also to a **Selector** interactor, and pass the first output of **Selector** to the open input of **Select**.
- Drag the **Selector** interactor with the middle mouse button into the control panel that exists for the **FileSelector**.
- Run the output of **Select** to an **Isosurface**. Set the *value* of the **Isosurface** to 100.0.
- Hook the output of the **Isosurface** into a **Color** module. Set the *color* parameter to "goldenrod" and the *opacity* parameter to 0.3.
- Add an **Image** module to render the colored isosurface. Execute Once.

After the first execution, look at the **Selector** interactor in the control panel. This interactor is data driven, since it examines the imported object, determines what fields it contains, then displays the field names for you to select which field to visualize. A different data file, with different field names, would create a different selection list. Thus, the data driven interactor automatically updates the field names to those contained within the imported data object. Many modules in OpenDX can be data driven.

Double-click on the name "Select which:" in the Control Panel to open the Set Attributes dialog for this interactor. Note the correspondence between the integers 0, 1 and 2 and the field names. The integers 0, 1, or 2 are the first output of the **Selector**, and the field names are the second output. The value passed to the **Select** module is actually one of the integers, but the actual field names will work with the **Select** module as well. The reason that either will work is because the OpenDX data model describes fields with both a name and a number, and the data-driven **Selector** captures both types of information.

- Use the view control dialog of the Image window to set the view to "Off Top", which is a good position from which to view the data set. Set the Projection to "Perspective" with a view angle of 30.0.
- Add a **ShowBox** module (from **Realization**) after the **Import** and **Collect** it with the current object before passing it to **Image** (Figure 10.3).

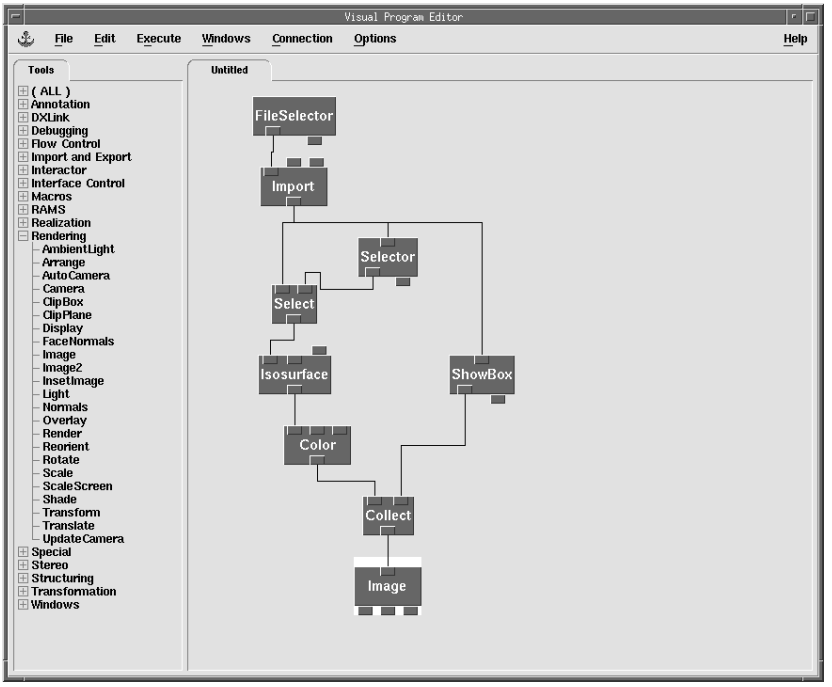


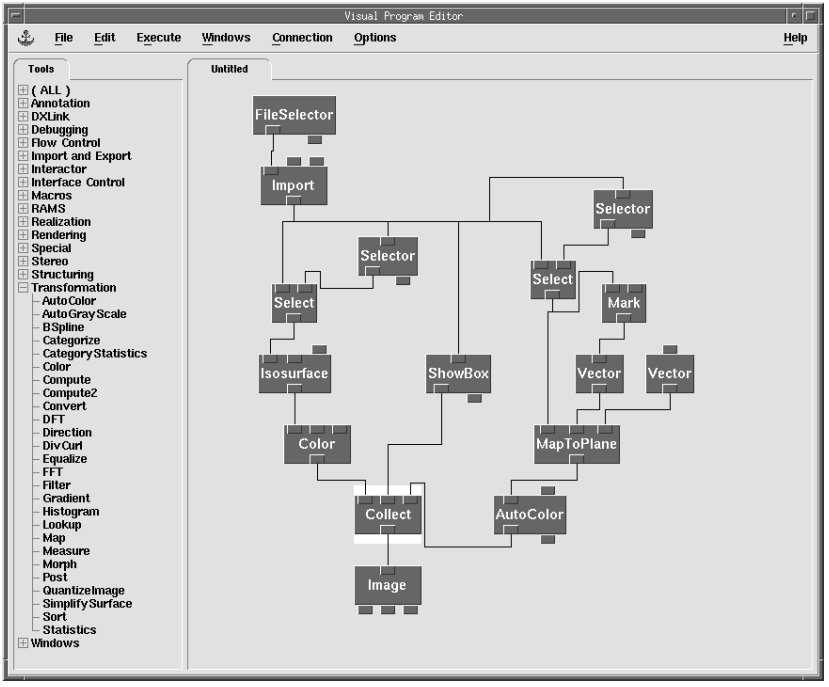
Figure 10.3
Current VPE

OpenDX provides various modules that can produce "slices" through a 3-D data set. Two modules in the Import category, **Slice** and **Slab**, can almost accomplish the task you want to perform. At first thought, **Slice** seems to do exactly what you want. However, neither of these modules actually create arbitrary slices through the data set; instead, they both create a subset of the input which can be depicted to appear as a "slice". Since you want to be able to view a slice from arbitrary planes throughout the data set, you cannot start by subsetting the data. Thus, the module you need for this visualization is **MapToPlane**.

- Copy the **Select** and **Selector** modules at the same time (this will make the new ones automatically connected), pass **Import** into the new **Select** and **Selector**, and drag the **Selector** into the existing Control Panel.
- Change the label on the first **Selector** interactor to "Isosurface Layer" and the label on the second **Selector** interactor to "Slice Layer".
- Place a **MapToPlane** (from **Realization**) below the new **Select** and wire the output of **Select** into the first input of the **MapToPlane**.
- **MapToPlane** needs two other inputs. Both of these inputs are three-vectors describing how to construct the plane. Wire a separate **Vector** interactor into each of these inputs and put the interactors into the existing control panel.
- **AutoColor** the output of **MapToPlane** and **Collect** this branch with the rest before Image.

- Set the MapToPlane point vector to {0, 0, -125} and the MapToPlane normal vector to {0, 0, 1}. Execute once.
- Data drive the **MapToPlane** point vector so that points located outside of the object cannot be selected. Wire a **Mark** module between the **Select** and the point **Vector**. Change **Mark**'s *name* attribute to "positions". Execute once, then play with the point vector interactor in the control panel to see the imposed limits (Figure 10.4).

Figure 10.4
Current VPE



- Change the Labels on the Vector interactors to Slice origin and Slice normal and set the limits on the Slice normal as -1 to 1 with an increment of .05. Make sure to set All Components.

It is difficult to see through the front of the semi-transparent exterior, so you next need to add a way to selectively clip the front of the object off for better viewing into the interior.

- Between the **Collect** and **Image**, add a **ClipPlane** (from **Rendering**). Add two new **Vector** Interactors to feed into the other two inputs of the **ClipPlane**.
- Set up the **Vector** Interactors similarly to the way you set up the two **MapToPlane Vector** Interactors.
- Set the initial **ClipPlane** Normal to {-0.5, 1.0, 0.0}. Execute once.

Notice that the **ClipPlane** clips all the objects coming out from the **Collect**. You don't want the bounding box to be clipped, so some parts of the network must be rewired.

- Unhook the **ShowBox** from the **Collect**. Place a new **Collect** between the **ClipPlane** and the **Image** and hook the **ShowBox** into that **Collect** (Figure 10.5).

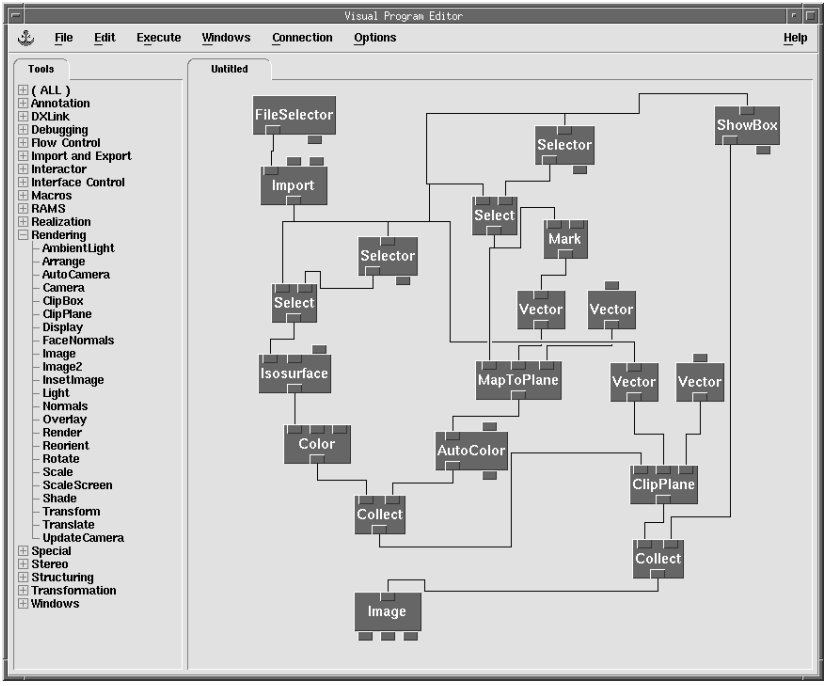


Figure 10.5
Current VPE slightly re-arranged.

- Now would be a good time to save this network. Save it as "mrb.net". If you play with the positions of the **ClipPlane** and the **MapToPlane**, you should be able to generate an image similar to the right hand Figure 10.2

You want the user to be able to turn on and off this clipping plane, so next look at the **Switch** module that will perform this function.

- Unwire the **ClipPlane** and the last **Collect**. Run the **ClipPlane** into the third input of a **Switch** module (from **Flow Control**). Wire the **Switch** output to the last **Collect**'s input. Run another output from the first **Collect** into the second input of the **Switch**.

The first input of a **Switch** module is the *selector* input. The *selector* input determines which of the inputs is passed through to the output. A value of 1 for the *selector* will pass the first input through, a value of 2 will pass the second input through, etc. A value of 0 for the *selector* of a **Switch** will pass a NULL object out of the output.

- Place a **Toggle** interactor near the **Switch** and wire its output to the first input of **Switch** (Figure 10.6).

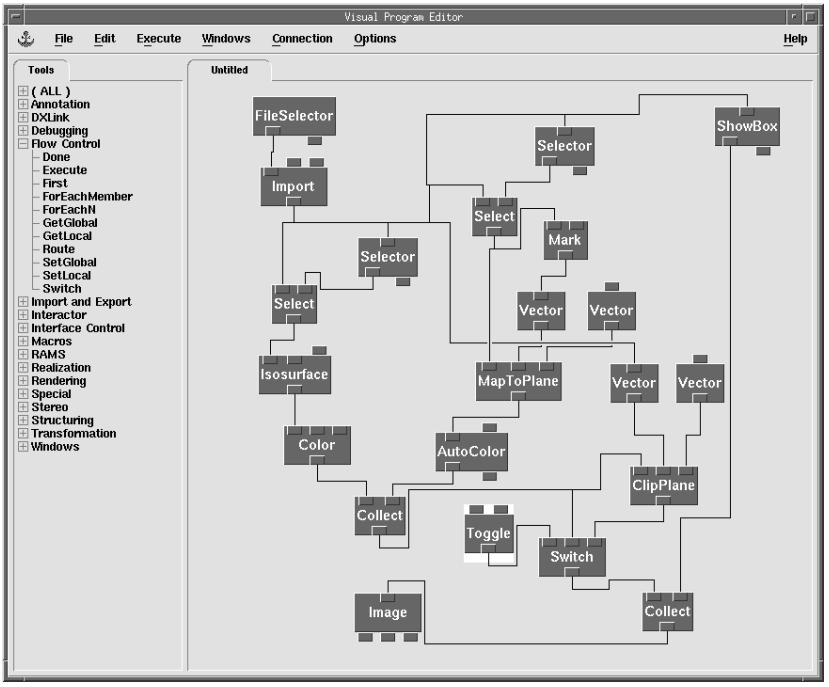


Figure 10.6
Current VPE

- Drag the **Toggle** into the existing control panel. Set the Attributes so that Button down (set) is 2 and Button up (unset) is 1. Change the Label to "Clipping Plane". Execute the program trying the toggle both set and unset.

Switch modules can be used programmatically as well as with interactors. It is possible to use a combination of **Inquire** and **Compute** modules to ask questions about data and thus control execution flow through various program branches.

Another module that controls data flow is the **Route** module, which works similarly to the **Switch** module. It differs in that it only accepts one input and the selector determines through which output to pass the resulting object. OpenDX will not execute any of the non-selected output branches of code.

The next requirement in the exercise is to create a second Image window with a slightly rotated image. The user should be able to select whether the rotated image is displayed or not.

- Below the last **Collect**, place a **Rotate** module (from **Rendering**) on the canvas. Below the **Rotate**, place a **Display** module (from **Rendering**) on the canvas. Wire the network with the output of the last **Collect** going into the first input of the **Rotate**, the output of **Rotate** being the first input of the **Display**, and the second output of **Image** being the second input of **Display**.
- Set the parameters of **Rotate** with *axis* set to "z" and *rotation* set to 90.0. Execute once.

You will see a second Image window appear with the objects rotated 90 degrees. This is what the exercise requires; however, you must make this **Display** window open and close under user control.

- Place a **ManageImageWindow** (from **Interface Control**) on the canvas. Place a **Toggle** interactor near the **ManageImageWindow** and wire the **Toggle** output into the second input of the **ManageImageWindow**.
- Open the **Display**'s Image window by clicking on the **Display** module and selecting Open Selected Image Window(s) from the Windows menu. Click on Change Image Name … in the Options menu of **Display**'s Image window. Type the name in as "Rotated Image".
- Set the *name* attribute of the **ManageImageWindow** to "Rotated Image". Execute the program trying it with the toggle button on and off.
- Change the label on the Toggle button to "Show Rotated Image".
- Try rotating the image in the Display window. Now try rotating the image in the Image window. Now try rotating the image in the Image window with the rotated image turned off.

If you watch closely, you should notice that OpenDX still renders both the Image and the Display. **ManageImageWindow** opens and closes the Image windows, but does not tell OpenDX to stop rendering. Rendering images that are not displayed is expensive in terms of both time and space, and should be avoided if possible. In order to make this network cease rendering the Display when it is toggled off, you must add a **Route** module.

- Place a **Route** module (from **Flow Control**) between the last **Collect** and the **Rotate** module. Wire the output of the last **Collect** into the second input of the **Route**, and wire the first output of the **Route** into the first input of the **Rotate**. Wire the output of the "Show Rotated Image" **Toggle** into the first input of the **Route** (Figure 10.7).

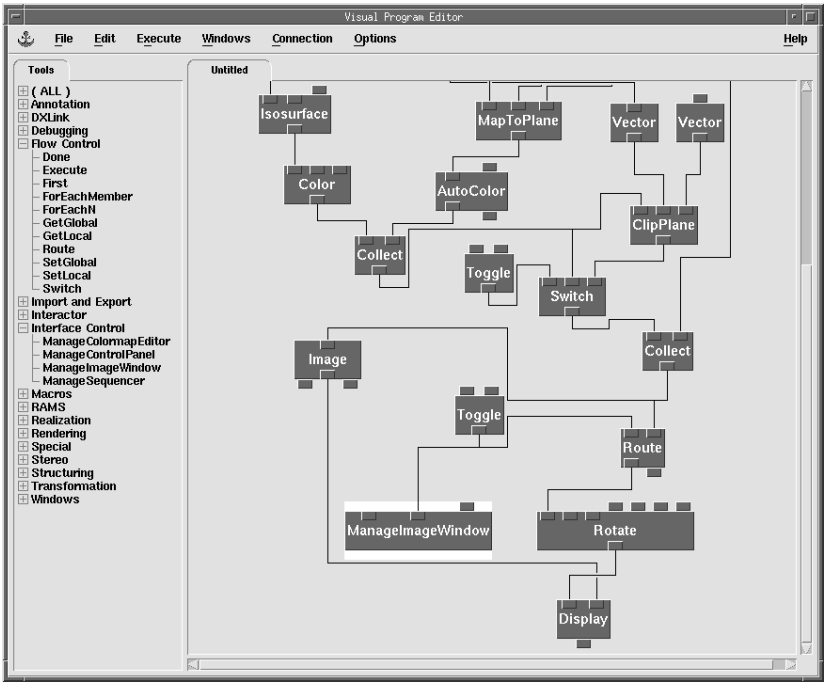


Figure 10.7
Current VPE

When the "Show Rotated Image" Toggle is off (set to 0), the **Route** will stop all execution of modules that lie downstream. However, when the **Toggle** is on (set to 1), **Route** will allow all modules that lie downstream of output 1 to be executed.

When presenting a program to an end-user, it helps to organize the control panels. There are some items that can be added to Control Panels to enhance their appearance.

- Open your Control Panel and manipulate all the interactors, stretch their lengths (ctrl-click-drag on a corner), change their style and orientation, and add separators.
- Now click on Dialog Style under the Options menu. This new appearance is how it will appear to end-users (Figure 10.1). Click Close on the Control Panel and answer No to the question of wanting to revert the style back.
- Save your network and exit OpenDX.

OpenDX can be used to deploy custom networks as full stand-alone programs. Once a network is written, it can be invoked as a command with a few extra command line options to become a custom visualization program.

- At the UNIX command line, type the following: dx -image -execute_on_change mrb.net. Also try the following: dx -menubar mrb.net.

Review

Route and **Switch** are two powerful flow control modules. They differ in that **Switch** always continues to pass information down the execution flow, but may sometimes pass a NULL value. **Route** however stops the flow of execution down branches of non-selected outputs. If a non-flowing execution branch (routed off) is an input to a valid execution branch, the valid branch does not execute. There is one exception to this rule using the **Collect** module, see the Reference manual for more information. Sometimes it is necessary to use a combination of Routes and Switches to reach a desired result. The other flow control modules, mainly looping, are discussed in a later chapter.

Other modules besides interactors can be data-driven. Throughout this book you have used modules that need bounding information. The data model allows modules to gather this information from the data structures as needed. Any module with minimum and maximum inputs can have a field with a 'data' component passed to it, and OpenDX will calculate the minimum and maximum.

As mentioned in the text above, the **Selector** uses integers for the first output and a string for the second output. Note that the first output of a **Selector** can be items other than integers.

There are other "Manage" tools that open and close control panels, including the **Colormap** editor and the **Sequencer**. These work in the same fashion as **ManageImageWindow**. When writing a network, it becomes useful to have a control panel containing specific information to pop-up when some condition becomes true. Note that if you close an Image window using **ManageImageWindow**, you must also route off execution of the **Image** tool to prevent it from reopening when Image executes.

OpenDX can also be used as a complete application development product. You can create a visual program and distribute it to be invoked as a system using the command line arguments or a simple script file. A developer can even encrypt networks and override the OpenDX copyright and splash screen to preserve property rights.

Display versus Image

In general, creating an image for display to the screen involves complex and time consuming rendering, involving converting a geometrical and color description of an object to a matrix of pixels. (An exception is the case where you already have an image, for instance the output of **ReadImage**. We will discuss that shortly.) To perform rendering, you need an object to render, and a camera to specify the viewpoint.

The **Image** tool, which provides direct interaction (rotation, zoom, etc.), takes as input the object to render. The **Image** tool derives the initial camera from the object itself, and afterwards deduces camera position from mouse movement. The **Display** module takes as input the object to render and an explicit camera to specify the viewpoint. The camera can come, as it did here, from an **Image** tool, or it can come from the **AutoCamera** or **Camera** modules, if you want to specify a precise viewpoint.

Note that if you input an existing image, such as one read from a file using **ReadImage**, you do not want to pass it to **Image**. Rendering an image is the same as rendering many little quads instead of simply displaying it to the screen. Instead, you should simply pass such images directly to **Display** without a camera.

Another module that performs rendering is (appropriately) the **Render** module. However, it does not display the image to the screen. It simply creates the image for further processing, such as filtering (using **Filter**), arranging it in a panel with other images (using **Arrange**), or writing to a file (using **WriteImage**). Once you have this type of processed image, you can use **Display** without a camera to show it.

A set of modules can be used to provide the same functionality as the **Image** module less user interaction. These modules would be a combination of **AutoCamera**, **Render**, and **Display**. For example, when creating large animations that will be written first to disk and subsequently converted to video, you have more programming control using **AutoCamera**, **Render**, and **WriteImage** than using an Image where control is interactive.

For advanced users, the **SuperviseWindow** and **SuperviseState** modules allow you to add extra functionality, such as to define your own interaction modes (e.g., if you don't like the supplied rotation operations, you can define your own) or create interactive panels of "arranged" images. The Supervise functionality is a complex topic; more information is available in the User's Reference under these two module names.

Control Panels

Since the end-user usually sees the control panels containing interactors as an end product, you can truly customize the control panel. OpenDX provides the developer with a large degree of flexibility. Control panels can:

- have data-driven interactors (autoconfigure based on data being viewed);
- allow for customizing the type, label, and placement of interactors;
- have multiple instances of interactors in one or more control panels;
- save and restore interactor settings;
- be placed in a hierarchy;
- have functional documentation; and
- be saved in "dialog style" for more seamless applications.

Make sure when delivering OpenDX networks with set values and nicely laid out control panels to include the .cfg file, because OpenDX stores the Control Panel information within these files.

You can take advantage of a few of the control panel's special features. You can:

- change size of interactors using control-mouse click drag on a corner;
- create multiline custom labels using backslash-n;
- change interactors to have vertical or horizontal orientation; and
- change the appearance of certain interactors.

11 Series, Categorical, and Scattered Data

Rationale

OpenDX calls data collected over time or produced in a time based simulation, series data. Series data can be in single file or multiple files. For series data in multiple files, the user must create a header that describes the series data and logically links the files together.

Categorical data are simply categories or bins that contain other data types. OpenDX uses "categorical" data to reduce the size of a component that contains duplicate values, strings or vector data. That is, in many circumstances OpenDX will detect repeated values in a component and place them to a single category.

Scattered data involve both data values and position values. OpenDX can import a data file that contains position descriptions as well as data values. This can be done using the general array importer, ImportSpreadsheet or converting the data file to a native OpenDX file.

You should complete the following three exercises and compare your results to the step-by-step instructions in this chapter. The exercise instructions contain some important information for working with series, categorical, and scattered data.

Exercise 1. Series Data

Use the Data Prompter to import the "amerSeries.bin" data file. Create a simple visualization to examine how OpenDX stores the imported data field. Use a Select and the Sequencer to animate auto colored frames from the series. Reduce memory usage by removing the Select and feeding the Sequencer as input into Import to select one of the series positions. Investigate how OpenDX internally structures the field importing in this fashion.

Exercise 2. Categorical Data

Import the data file "carMiles.txt" as spreadsheet data and plot the mean of the miles per gallon and the mean horsepower for each make and then each model. Label each plot accordingly.

Exercise 3. Scattered Data

Given the data file "tmpScattered.txt", import the data and locations into OpenDX using both the general array importer and the **ImportSpreadsheet** module. Note that this file has values that should be used to define the 'positions' component for the imported object. Depending on the layout of the file, there are two possible solutions for importing the data. The **ImportSpreadsheet** can be used here, because the data are in columnar spreadsheet style, or you could use the general array importer if the data are in either columnar style or block style. Auto glyph and auto color the temperature, auto glyph the city name and display the resulting image. Be aware of any 'connections' components.

Step-by-step instructions for Exercises

Instructions for Exercise 1

- Start the Data Prompter and use Select Data File... from the File menu to select "amerSeries.bin". This file is a regular array.
- Browse the data to see the header (most of the file is binary as you can see). Click on the Single time step button so that it is disabled, since the header says there are 10 steps.
- Click on the Describe Data... button and fill in the prompter with the information from the header, see Figure 11.1.

Figure 11.1 Completed Data Prompter Window

If at any time certain options that you need are not visible in the prompter, select the Full Prompter option under the Options menu.

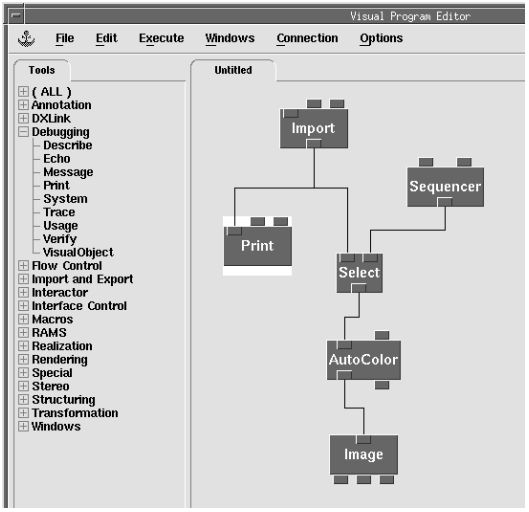
- Save this header as amerSeries.general. Try the Visualize Data... option and you should see a representation of the North American continent. You can use the Sequencer to cycle through the series.
- Exit out of the "Data Prompter" and start the visual programming editor. Place an **Import** module on the canvas and import "amerSeries.general".
- Place a **Print** following **Import**, specifying the *option* "o". Execute.

Look at the message window. Notice this object is a Series Group and lists the number of members.

- Place a **Select** module (from **Structuring**) below **Import** and connect the output of **Import** to its first input. Connect the output of the **Sequencer** to the second input of **Select**. Set the **Sequencer** to run from 0 to 9.

- Place an **AutoColor** following **Select** and run the output of **AutoColor** into an **Image**. Play the **Sequencer** (Figure 11.2).

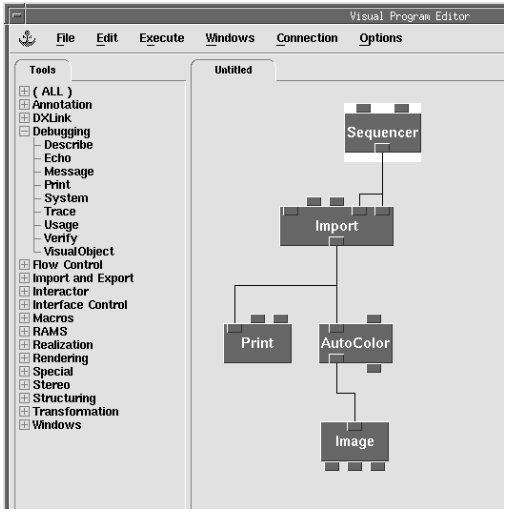
Figure 11.2
Current VPE



An alternative to importing the entire series and selecting the desired time step is to import the desired time step one at a time. The next step demonstrates this technique.

- Remove **Select**, and expose the *start* and *end* inputs for **Import**. Connect the output of the **Sequencer** to both. Wire the **Import** directly to the **AutoColor** (Figure 11.3). Play and observe the result.

Figure 11.3
Current VPE



For a large data set, it is often desirable to import and store data for a single step, instead of importing the entire series. That is, if OpenDX stores the entire series object in memory, this can limit how much memory is available for subsequent processing.

- Save the program as "series.net." Quit OpenDX.

Instructions for Exercise 2

- Look at the file "carMiles.txt" using the Unix command *more* or a text editor.

Note the layout of the file and count the number of header lines at the top. The file is also tab-delimited. This file contains data about a variety of makes and models of cars. It gives information such as miles per gallon, volume, horsepower, and weight.

- Start a Visual Program Editor session (dx -edit). Place an **ImportSpreadsheet** (from **Import and Export**), and fill in the *filename* with "carMiles.txt" and the *delimiter* with "\t". In order to skip the 6 lines of header, expand the configuration dialog and scroll down to fill in *headerlines*.
- Print with *option* "rd" the output of **ImportSpreadsheet**. You will see a number of components (note that none of them are the 'data' component). Many of the components are of type string.

Modules like **AutoColor** and **Isosurface** do not know how to deal with "string" data. This is where the Categorize module is useful; it can transform "string" data to numerical data through mapping or *binning*.

- Place a **Categorize** (from **Transformation**) after **ImportSpreadsheet**. Pass the output of **ImportSpreadsheet** to the first input, and set the second input to the component name "MAKE". Now print the output of **Categorize**.

Look specifically at the 'MAKE' component. Note how it is now an array of numbers. These numbers reference into a new "MAKE Lookup" component named after the categorized component. The lookup component is a sorted array of unique string values from the original 'MAKE' component.

Categorize thus takes a list of values, finds all unique values and creates an indexed sorted list from them. The component is then changed to become a list of values that references the lookup table. Like the "C" programming language, OpenDX references the index into the lookup table starting at 0.

The **ImportSpreadsheet** module can categorize during import. It can categorize any component you name, or for convenience, categorize all "string" components.

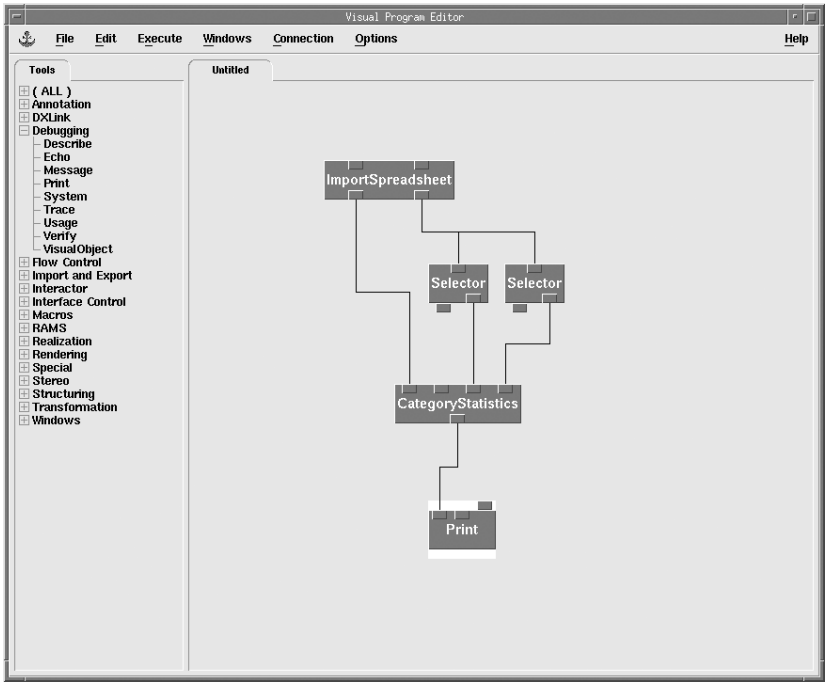
- Open the configuration dialog box for **ImportSpreadsheet**, and find the *categorize* input. Use the ... button to find the correct option so it categorizes all the string data, i.e. select "allstring".
- Now you can remove the **Categorize** module. Print the output of **ImportSpreadsheet**.

Note how both the "MAKE" and "MODEL" are both *binned*. By converting the string data to numerical data, you can now use these columns with most of the modules in OpenDX.

There is a useful module called **CategoryStatistics**, which allows OpenDX to relate various data items within a categorical data field.

- Place **CategoryStatistics** (from **Transformation**) below **ImportSpreadsheet**, and pass the first output of **ImportSpreadsheet** into its first input.
- Set the *operation* input of **CategoryStatistics** to "mean" and unhide the hidden parameters *category* and *data*.
- Place two **Selectors** (from **Interactor**) on the canvas between **ImportSpreadsheet** and **CategoryStatistics**. Wire the second output of **ImportSpreadsheet** to each **Selector**. Wire the second output of one **Selector** to the *category* input of **CategoryStatistics** and the second output of the other **Selector** to the *data* input.
- Wire the output of **CategoryStatistics** to the **Print** and execute once to initialize the **Selectors**, Figure 11.4.

Figure 11.4
Current VPE



- Place both **Selectors** in one Control Panel. Change the "category" **Selector** to "MAKE" and the "data" **Selector** to "HP". Execute once.

Look at the message window. Note that you now have a 'data' component and a 'MAKE lookup' component. **CategoryStatistics** performs functions based on each category. In this case, you named

the category 'MAKE' and asked for an *average* of the horsepower. **CategoryStatistics** works through the 'MAKE' component averaging all the horsepowers per unique 'MAKE' value. This calculates the average horsepower per make of automobile. Since data exists, OpenDX can create a plot.

Plots require a data set with one-dimensional positions and a data value. **Plot** uses the position as the x-axis and the 'data' component value as the *y*-axis, which is exactly what you have after the **CategoryStatistics**.

- Pass the output of **CategoryStatistics** into **Plot** (from **Annotation**) and the output of **Plot** into **Image**. Execute once.

The image does not look bad, but labels on the x-axis are imperative to allow an observer to know what the plot represents. If you pass in a list of strings to **Plot**, it will interpret them as labels for the axes. The 'MAKE lookup' component contains the list of strings you need as axes labels.

- Extract the 'MAKE lookup' component using **Extract** (from **Structuring**) and pass it to the *xticklabels* input of **Plot**. Change the *labelscale* of **Plot** to 3.0. Execute once.

Remember that OpenDX creates the lookup tables during the categorize step in **ImportSpreadsheet**. For each of the string data components, there is a lookup table that corresponds to the string component. But after the **CategoryStatistics**, only one of the lookups is passed through.

- Using the Selector, change the "data" from "HP" to "MPG". Execute once.

Notice how the plot changes to reflect the average miles per gallon. You may need to reset the image in order to get the plot re-centered. This is due to the fact that the camera for the image is set to the previous plot's center. Automating the rest is left as an exercise for the reader.

- Using the Selector, change the "category" from "MAKE" to "MODEL". Execute once.

Note that you get an error at this point because the available "lookup" table is now the 'MODEL lookup' and not the 'MAKE lookup'. It would be easy to change the value set for the **Extract** module to 'MODEL lookup', but this is a problem if you need to change this value each time the selector is changed. You can programmatically tell OpenDX which component to extract by using the **Selector** in combination with the **Format** module.

The **Format** module works like a printf in the "C" programming language. First you pass a template describing what the resulting string should look like, placing %s where a string substitution should take place, %d where integer substitution should take place, and %f where floating point number substitution should take place. In this case you also need to add the word "lookup" after the value you get from the **Selector**. Thus your template will be "%s lookup" where %s is the string you get from the Selector.

- Place a **Format** (from **Annotation**) above the **Extract** module. Wire the second output of the category **Selector** into the second input of the **Format** module. Set the *template* of **Format** to "%s lookup". Wire the output of **Format** to the second input of **Extract**, Figure 11.5.

Note that this will substitute the string passed from the **Selector** into the template at the "%s" position. So when "MAKE" is selected, the output of the **Format** is "MAKE lookup". When "MODEL" is selected, the output is "MODEL lookup".

Execute the program selecting different categories and data options. Note that if you select a numerical data component for the categorical data an error will occur. This is due to the fact that only the string data had been categorized in the **ImportSpreadsheet**. It is possible to categorize numerical data as well.

- Save the program as "category.net".

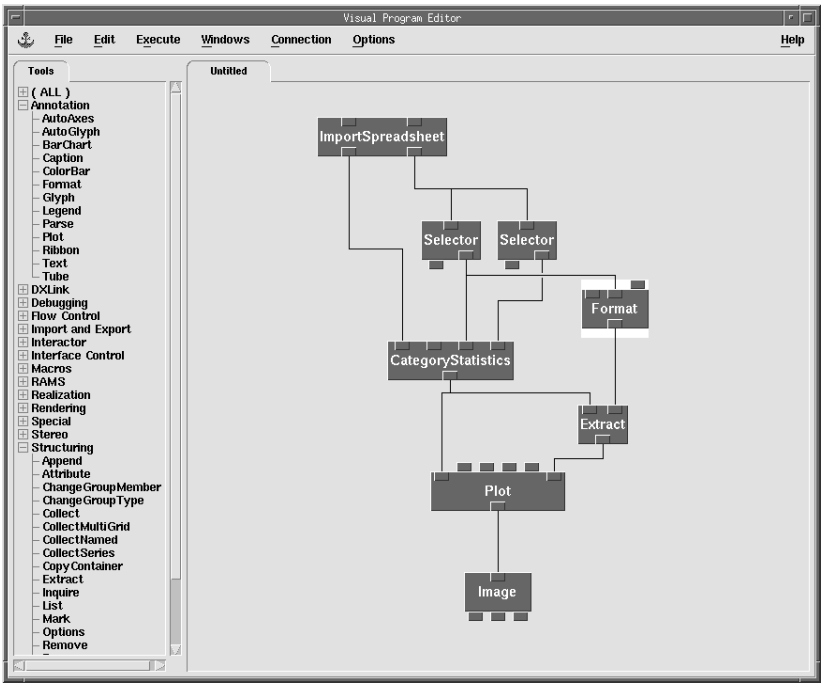


Figure 11.5
Current VPE

Instructions for Exercise 3

- Start the Data Prompter, select the data file "tmpScattered.txt", and click on the Grid or Scattered file. Browse the data file to look at its layout.
- Next to Grid type, click on the scattered type (the button with random dots on it). Change the number of variables to three (the temperature, city and state). Click the Positions in data file button and change it to Dimension: 2. Change the Data organization to Columnar.

At this point, you have almost completely described the data file; however, you need to give the exact layout of the file.

- Click the Describe Data... button. Fill in the information for the Header (1 line) and the number of points (56). The Data format is Text so that will not change.

The order of the variables is important in the Field list. In the file, the variables are listed as City, State, temperature, latitude, and longitude. Thus you must reorder the field list and rename the variables in the process. Note the data prompter uses the keyword "locations" as the name of the positions in a field.

- Use the Move field buttons to arrange them in order as {field0, field1, field2, locations}. Rename "field0" to "city" and make it a string of length 15. Change "field1" to "state" and make it a string of length 2. Rename "field2" to "temp".

The general array importer has some strict parsing algorithms for dealing with files. OpenDX assumes data are separated by a number of characters. Spaces and tabs both separate data. So telling OpenDX that a string is of length 18 does not mean it will always read in 18 characters. If OpenDX locates a separator within a character string it will assume that the next set of characters belong to the next field. Thus in this file, you will note that city names that are two words have an underscore in place of the space. This is not a problem with **ImportSpreadsheet**. In fact, you will bring in the city name and state together as one variable when using the **ImportSpreadsheet** later.

- Save the file as "tmpScattered.general". You may quit the prompter at this time.
- Start a Visual Program session and **Import** the *general* file you just created.
- Place a **Select** after the **Import** and select the "temp" field.
- Place an **AutoGlyph**, **AutoColor**, and **Image** in that order on the canvas after the **Select**. Hook the network up in order and execute once (Figure 11.6).

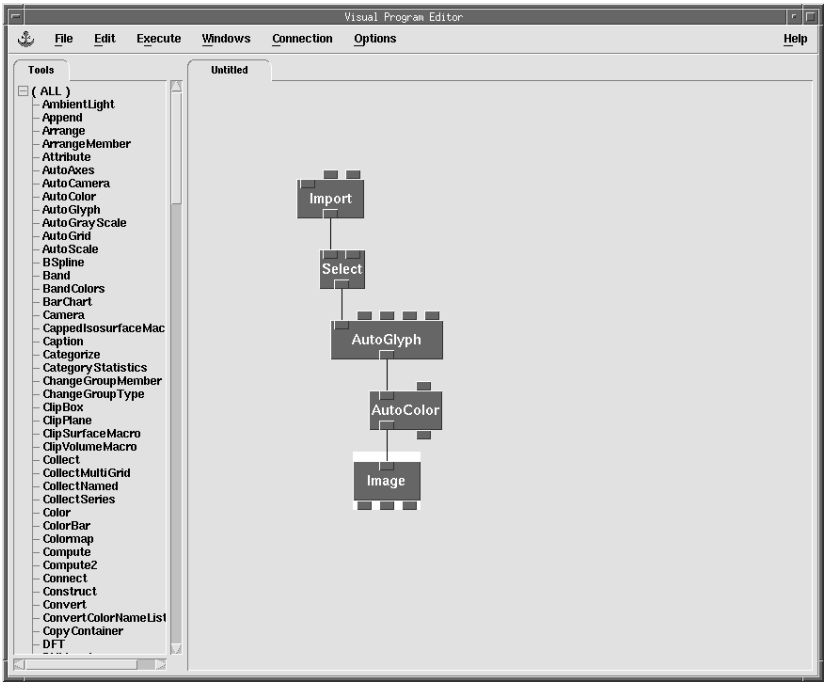


Figure 11.6
Current VPE

At each point on the map, display the name of the town associated with its temperature.

- Place another **Select** below the **Import** and select "city".
- Pass the "city" field to an **AutoGlyph** and **Collect** it with the temperature glyphs before the **Image**. Change the glyph *type* to "text". Execute once.
- Zoom in and out to investigate how the text auto glyph works.

Note that the image seems to be reversed; however, this is correct for this data set since the longitudinal numbers are stored as positive, not negative, numbers (longitude is increasingly "negative" as you move west in the Western Hemisphere from the origin at Greenwich, U.K.). In order for the image to look like what you want, you must use a **Transform** module right before the **Image**.

- Place a **Transform** (from **Rendering**) module between **Collect** and **Image**, Figure 11.7. The *transform* to apply is [-1 0 0][0 1 0][0 0 1]. You will have to reset the image after executing.

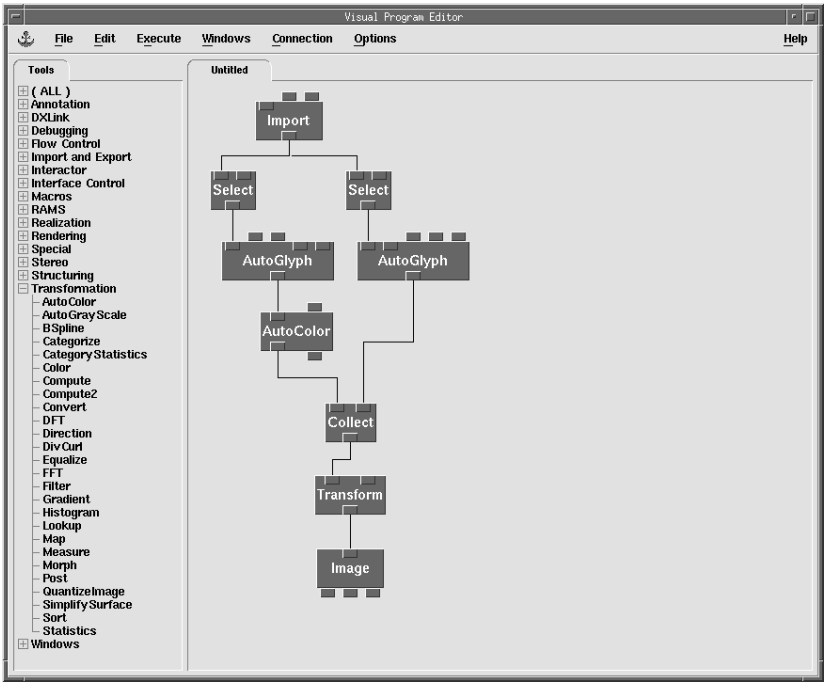


Figure 11.7
Current VPE

For the rest of this visualization, make sure to apply the **Transform** to all data before sending it to **Image**.

- Play with the shape, scale, and ratio of the glyph until a desirable result is obtained.

To understand how **ImportSpreadsheet** and the general array importer differ, produce the same image with a network that starts with **ImportSpreadsheet**.

- Without deleting any part of the current network, place an **ImportSpreadsheet** to its right. Import the data file "tmpScattered.txt" and change the *delimiter* to "\t".

All of the columns become components to one field from an **ImportSpreadsheet**. None of these components are the 'positions', thus you must use modules to construct a 'positions' component.

- Wire **ImportSpreadsheet** to two new **Mark** modules. Mark "Long" with the left module and "Lati" with the right module.
- Run both of the **Marks** into a **Compute** module. Set the **Compute's** *expression* to [a, b].

If the Longitude and Latitude are not correctly input into the **Compute**'s "a" and "b" tabs, then the positions will be transposed as [y, x] and the image will be incorrect.

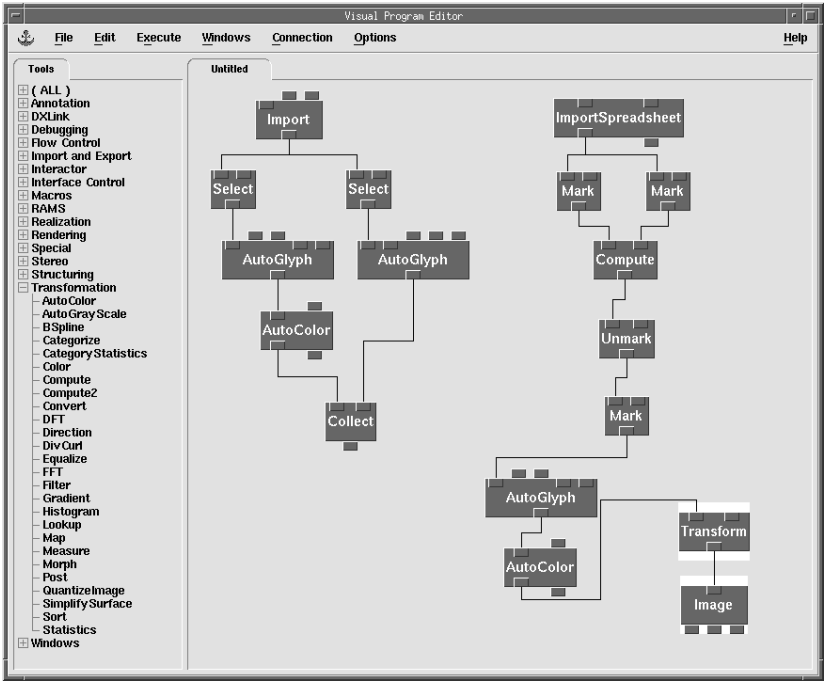
- Place an **Unmark** below the **Compute** and unmark "positions".

This will create a new 'positions' component overwriting the old with the positions becoming the files Long/Lat. The field is now very similar to what was imported with the general array importer; except

all the variables are components to a field instead of a group of fields. Variables used as components must have their own name, but to be used with modules such as **AutoGlyph**, a variable must be marked as the 'data' component.

- **Mark** the "Tmp" component of the result from **Unmark**.
- Copy the **AutoGlyph** and **AutoColor** from the previous temperature branch of the network, and move the **Transform** and **Image**, wiring the network as in Figure 11.8. Execute once.

Figure 11.8
Current VPE



The resulting image should be identical to the one produced for the temperature glyphs earlier. Just like in the other network, add the city name to the temperature glyphs. The field after the **Unmark** can be used to accomplish this.

- Wire the **Unmark** into another **Mark**, and mark the "City" component.
- Copy the "text" **AutoGlyph** and the **Collect** over and wire the network similar to the way it was done previously (Figure 11.9). Execute once.

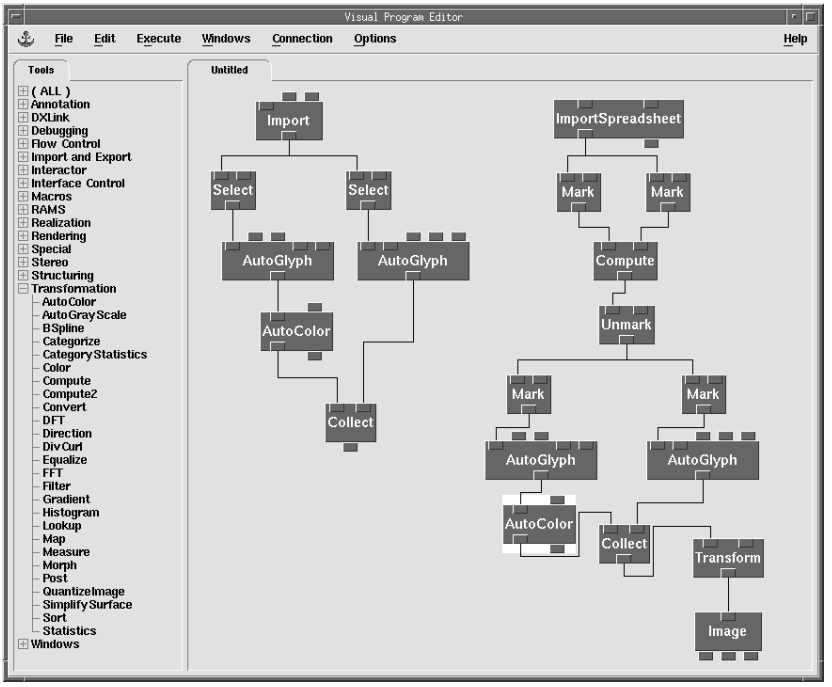


Figure 11.9
Current VPE

Look at the two networks and note the differences. Try removing the **AutoGlyph** for the temperature and discover what happens. See if you can explain it to yourself.

Review of Chapter 11

You can import series data as a single time slice or as one object and then use **Select** to select one time slice. The **Sequencer** is a valuable tool to animate series data. OpenDX designates series data as a special group. The native OpenDX file format can handle series data with ease.

ImportSpreadsheet is a quick way to import spreadsheet type data into OpenDX. Each column imports as a separate component to one field. The column header is the name of that component. You select fields in a group using the **Select** module whereas to select components in a field use the **Mark** or **Extract** modules. After using **ImportSpreadsheet**, you typically need to **Mark** a column as 'data'.

Often for scattered type data, columns in a spreadsheet are the positions. You can apply a **Mark**, **Compute**, and **Unmark** sequence of instructions to create positions from the components.

If OpenDX imports data as "1-d" using the **ImportSpreadsheet** module, then the positions become the indices (row numbers) of the imported file. If OpenDX imports data as "2-d" then the output field will be a C x R grid, where R is the number of imported rows, and C is the number of imported

columns. The field will have a single data component that contains all the values in the imported rows and columns.

Categorize converts a component of any type to an integer array that references a newly created 'lookup' component. The 'lookup' component is a sorted list of unique values of the original data that:

- reduces the size of a component that contains duplicate values;
- converts string or vector data into "categorical" data; and
- allows the detection of repeated values in a component.

Categorize uses the smallest possible memory block to store "categorical" data, e.g. it uses Byte if the category has 256 or less unique values.

CategoryStatistics works on *binned* data. It

- creates a new 'positions' component representing the categorical indices;
- creates a new 'data' component containing the requested statistics; and
- performs a calculation on the named data per category (bin).

Plot always expects a field with a one dimensional positions component (this will become the x-axis,) and a one dimensional data component (this becomes the *y*-axis.)

Looping and Probing 12

Rationale

The programming interface of OpenDX is based on a more or less traditional data flow architecture, but also includes extensions that provide both conditional execution tools (**Route** and **Switch**) and explicit looping tools (**ForEachMember**, **ForEachN**, **Done**). It is important to understand these control flow constructs since they provide extended functionality that is used in many examples. OpenDX also includes modules **Get** and **Set** that allow the user to explicitly control data caching, which is important when loops are used.

OpenDX allows you to interact with and investigate the data set within the Image window. Using the special **Probe** and **Pick** tools, the end-user can interactively select or position points in space to gather information about the rendered objects.

The exercises in this chapter illustrate complex data and flow control. You may try to complete the two exercises on your own or may find it beneficial to follow the provided instructions. In either case, you should be sure to read all of the accompanying text. It contains a great deal of explanation about how OpenDX handles looping and interactive picking.

Exercise 1. Looping and Macros

Your first task is to use the flow control modules to create a loop that echoes the values of 0 through 10 to the message window. Later, you will extend this functionality by stopping the loop with the **Done** module at a user-specified value. The next task is to start a new visual program and import the "amerSeries" data. Then, accumulate the data over the grid for all ten-time steps. Next, create and use a macro to view only the final accumulation. Create another network that uses an "until done" loop to count to 50. Finally, examine the implications of using interactors with the **Done** module.

Exercise 2. Probes, Picks, and Text Glyphs

Use the data file "sealevel.bin" from Chapter 3 to create an autocolored rubbersheet. Then use the mouse with the probe and pick to show the data values (elevations) at points on the earth's surface.

Step-by-step instructions for Exercises

Instructions for Exercise 1

- Start the Data Explorer VPE by typing dx -edit.
- Place a **ForEachN** module (from **Flow Control**) on the canvas. Open its configuration dialog box and set the *start* parameter to 0 and the *end* parameter to 10. Close the configuration dialog box. Attach an **Echo** module (from **Debugging**) to the first output of **ForEachN** and execute once.

Notice that the **Echo** module executes 11 times, printing 11 values, although you only executed the program once. Execute again; the same thing should happen.

You have just seen that a single loop (with the start and end values specified in **ForEachN**) completes as a single execution in Data Explorer. This is important to remember for future use, especially for use with interactors and macros.

OpenDX provides additional loop control modules. For example, the **Done** module provides an early exit from a loop.

- Place **Done** (from **Flow Control**) on the canvas. Connect the first output of the **ForEachN** to a **Compute** (leave the connection to **Echo** as is) and enter the expression "a > 5 ? 1 : 0". Connect the output of **Compute** to **Done** and execute.

The expression "a > 5 ? 1 : 0" is special syntax (borrowed from the "C" programming language) for an "if <condition> then <result1> else <result2>" statement. This particular expression reads as "if a is greater than 5 then return 1 else return 0." Thus, whenever the first input "a" is greater than 5, the **Done** module directs the loop to stop. In this case, the message window prints the values 0 through 6 because the first value greater than 5 is 6. Using **Compute** and **Done** modules together in this fashion allows you to create conditional looping.

Next, consider a more interesting looping example with the series data from the previous chapter.

- Clear the canvas using File->New. Place an **Import** module on the canvas and import the series data. The general file should be named "amerSeries.general"

Remember that the result of this **Import** is a series group with 10 different fields, each representing a time slice. If you wish to verify this, place a **Print** after the **Import** to view the group's structure.

- Place a **ForEachMember** module after **Import**. Wire them together.

ForEachMember consecutively selects each field from the series, starting at time zero and finishing when the last field is selected.

- Wire **ForEachMember**'s first output to an **AutoColor**, and connect the output of the **AutoColor** to **Image**. Execute once.

Notice that the visual output is similar to the example in the previous chapter, but the **ForEachMember** drives the selection of the steps. The **ForEachMember** provides no control over

playing, stopping, and reversing the selection with a single click of a button. Thus, it may not seem as useful as the **Sequencer**. However, explicit looping has some powerful applications that are demonstrated in the remainder of this exercise.

Suppose you want to sum all the series members together and make an image of the accumulated total. In addition to **ForEachMember**, OpenDX has a pair of modules that accumulate data while the loop runs. Assume that what you are trying to accomplish is summarized in the following pseudocode.

```
sum = 0
i = 0
while i < number_of_members
      i = i + 1
      sum = sum + the ith member  //Add each point in the field
end while
```

The **Compute** module adds data in the following way. If a field is one of the inputs to **Compute** and a scalar is added to the input, the scalar is added to every point in the field. If two fields (with the same number of data values) are input into **Compute** and added together, **Compute** performs a point-wise addition of the fields.

The **GetLocal** and **SetLocal** modules store intermediate results into a cache. These can be used to implement the variable sum in the pseudo-code above. In each iteration of the loop, **SetLocal** puts the intermediate sum value into cache; **GetLocal** retrieves this value out of the cache for the next addition.

- Place a **GetLocal** and a **SetLocal** (both from **Flow Control**) on the canvas. Add a **Compute**. Connect the second output of **GetLocal** to the second input of **SetLocal** to link them as a pair (so that **GetLocal** knows which cache to use). Connect the first output of **GetLocal** to the second input of **Compute**.

Remember that **GetLocal** provides the current value of "sum" to **Compute** as the second input "b" and **Compute** adds the current series member to it.

- Connect the output of **ForEachMember** to the first input of **Compute**, and enter the appropriate *expression* to compute the sum (a+b). Place the new sum into the cache by connecting the output of **Compute** to the open input of **SetLocal**. Initialize the output of **GetLocal** (the equivalent of setting sum=0 above) by entering 0 as the *initial* input of **GetLocal**. Execute once.

OpenDX produces the same result as before because it images only the original data values, not the values stored in the cache.

- Copy the **AutoColor**/**Image** pair and wire the output of the **Compute** to the first input of the new **AutoColor** (Figure 12.1). Execute once.

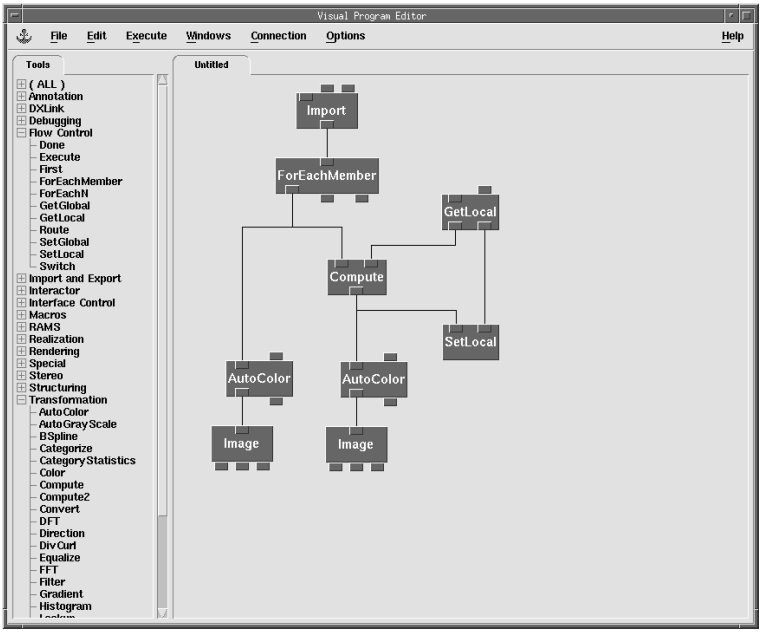


Figure 12.1
Current VPE

OpenDX now displays two images: the original series data, and the accumulation of the series. Compare these images to see if the accumulation occurs.

Whenever a loop appears in an OpenDX program, it raises the question of what part(s) of the program is interpreted as being in the loop, i.e. what part(s) should be repeated for each loop value? OpenDX considers ALL modules at the level of a looping construct to be in the loop. Thus if you want to have only certain modules of the program executed iteratively, you must create a macro to encapsulate the loop components.

- Delete the **AutoColor**/**Image** pair wired directly from **ForEachMember**.
- Select the set of tools **ForEachMember**, **GetLocal**, **SetLocal**, and **Compute**. Do this by rubberband selecting, or by shift-clicking on each module.
- From the Edit menu, choose Create Macro. Enter the *Name* LoopExample for this macro. Enter the *Filename* loopexample-macro.net. Press OK.

Note that the macro name is the name placed on the module box, whereas, the filename is the name of the small network as stored on disk. Although two separate and distinct names are used, it is a good idea to make the two match as closely as possible. When you define part of a network as a macro, all of its selected components collapse into one tool with the macro name. OpenDX also creates a new category in the Categories list named Macros, and within this category, it lists all currently defined macros.

- Execute once.

Notice that the image only displays the final result, not the intermediate results of all ten of the separate additions.

The important thing about putting looping tools inside macros is that the macro only returns the result, and does that only when the loop finishes. To see the intermediate steps, you must place the looping tools (**ForEachN** or **ForEachMember**) at the "top" level visual program, not in a macro.

Macros allow you to create subnets for repeated transformations, simplify your programming interface, and add customized tools to the programming toolbox. In many occasions, you may also have access to other programmers' macros, which can dramatically shorten the amount of time it takes you to develop a sophisticated program. Once a macro exists, you can also edit it as needed to produce your own customized version.

- Double-click on the macro. A configuration dialog box will appear, just as for any other module. To edit the macro, select the macro tool and choose Open Selected Macro from the Windows menu.

Note that the new VPE window contains all the tools in the original network, plus one **Input** and one **Output** tool at the start and end of the network, respectively. You can configure the **Input** and **Output** modules to define more appropriate input and output names for the macro. Open the configuration dialog box for an **Input** module and change the *name* parameter from input_1 to something that describes the type of input data (e.g., series). You can change the *output* name in a similar manner. After editing information in the **Input** or **Output** modules, the macro must be saved and reloaded in the original network for the changes to take place. After changing anything within a macro, remember to save it by selecting Save Program from the File menu of the macro's window.

OpenDX does not allow all the available modules to be included within macros. Modules that provide user interaction, such as **Image**, **Sequencer**, **Colormap** or any of the interactors cannot be included within a macro. Creating a macro can also be difficult if you are uncertain about exactly what the macro should do versus what the main program using the macro should do. In such a case, it is easier to initially include all elements in the main program, then create the macro for a subnetwork as shown in the above steps.

- Close the macro. Save your network as looping.net.

Up until this point in this exercise, all of the looping has been done with a "For" module, either **ForEachN** or **ForEachMember**. OpenDX can create loops without one of these modules present. For example, whenever you place a **Done** module on the canvas, you create an "until done" loop. In the following steps, you create this type of loop and investigate its implications.

- Clear the canvas. Place a **Done**, a **GetLocal** and **SetLocal** pair, and two **Computes** on the canvas.

- Wire the second output of **GetLocal** to the second input of **SetLocal**. Wire the first output of **GetLocal** into the first input of the first **Compute**. Wire the output of the **Compute** to the first input of **SetLocal**. Set the *expression* of the **Compute** to be "a+1". Set the *initial* value of **GetLocal** to "-1".

This set up creates a loop that starts with an implicit counter of -1 , adds 1 to the counter at each iteration, and repeats. Thus, the value available after the compute starts at 0 and increments forever.

- Wire the output of the first **Compute** to both an **Echo** and to the second **Compute**. Set the second **Compute**'s *expression* for the test: "if the input is greater than 50, output 1, otherwise output 0", i.e., "a>50?1:0".
- Pass the output of the second **Compute** to **Done** (Figure 12.2). Execute once.

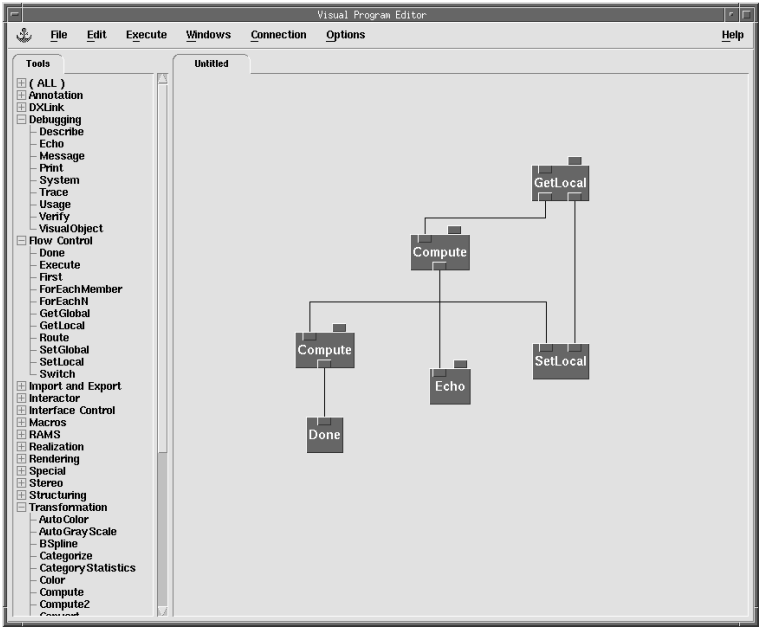


Figure 12.2
Current VPE

Open the message window and determine if this is the expected result. You should see the numbers up to 51 echoed, at which point **Done** terminates the loop. This is the basic until loop. At times you may want to define a continuous loop to allow an end-user to explicitly end the loop by clicking on an interactor.

- Place a **Toggle** interactor on the canvas and pass its output to the **Done** module. **Toggle** by default outputs 1 if pressed in, 0 otherwise. Execute once and experiment with pressing the **Toggle**. To stop execution use End Execution from the Execute menu.

From the execution result it appears that the **Done** module is working incorrectly because when you press the **Toggle** button the loop continues to execute. Actually, the module is working correctly, but

there are process interaction aspects to consider. Because OpenDX runs as a client-server process, the executive only gets the value for any interactor once, at execution launch. Once the network is running, the executive ignores interactors, including those whose values may have changed. To provide a more dynamic client-server interaction you would have to write your own user interface.

Instructions for Exercise 2

- Import the sealevel data from Chapter 3 into a new visual program. Create an automatically colored rubbersheet of the topography and use **Image** to display it.

Refer to Chapter 3 if you can't remember how to do this.

Suppose you want to use the mouse to select a point in the image and view its data value. The modules **Pick** and **Probe** allow you to select a position in the Image Window. Once you have "picked" or "probed" correctly you can use normal OpenDX facilities to display characteristics of the point.

- Place a **Probe** module (from **Special**) on the canvas. Connect the **Probe** to a **Print** module with the *options* of "rd".
- Open the View Control dialog from the Image Window and set the Mode to Cursors. Double-click on the terrain in the Image window. A cursor (green dot) will appear. The user can drag the dot to any position within the bounding box of the object. Execute once and look at the output of **Print** in the message window.
- Go back to the Image window and drag the little dot upwards. Execute once.

Notice that the values printed in the message window are the position of the dot, relative to the positions of the object. For example, if you drag the dot up, the second value (y) of the position display increases.

Suppose you want to display the data value at the probed point. **Probe** generally returns a vector value in three-space, but the original data is only two-dimensional. Thus, you must convert the position to 2-D space. You can then use **Map** to acquire the data value at that point. However, a single vector cannot be used as an input for **Map**, which requires an object as its input. You must use a **Construct** module to take a vector as input and construct an appropriate object as an output.

- Place a **Compute** between the **Probe** and the **Print**. Set the *expression* of **Compute** to [a.x, a.y]. Execute once and observe the output in the message window.

This removes the third dimension from the vector.

- Place a **Construct** (from **Realization**) between the **Compute** and the **Print**. Wire the output of the **Compute** to the *origin* input of the **Construct**. Execute once and observe what object **Construct** creates.

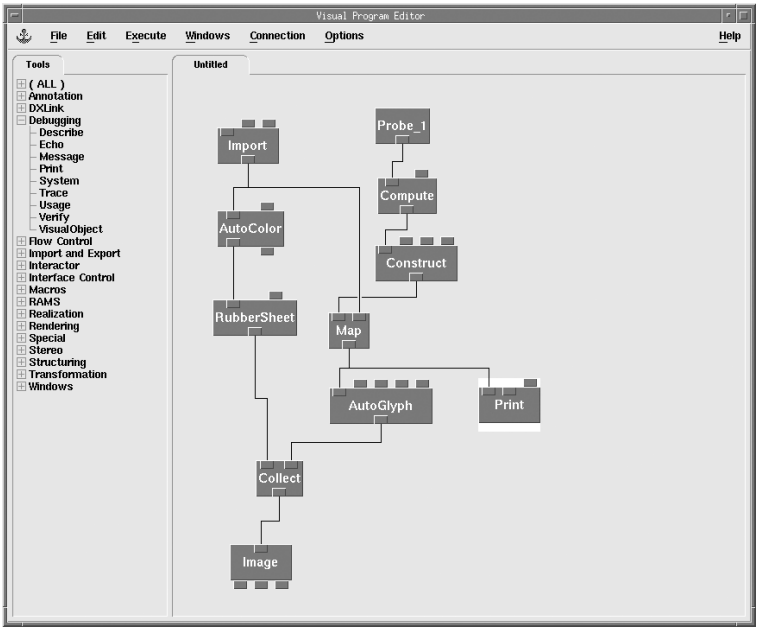
The output of **Construct** is a field with two components. The first component is 'positions', which is the two-vector you passed in as the origin. The other component is the 'box' derived component. Map can now use this field with the original data set to locate an appropriate data value.

- Wire the output of the **Construct** into the first input of a new **Map** module (from **Transformation**). Wire the output of the **Import** into the second input of **Map**. Use the **Print** module to print the output of **Map**.

The field from **Map** now has a data value assigned for the position from the **Probe**. You can use **AutoGlyph** to produce a text glyph that displays the data value at the location of this new field.

- Wire the output of **Map** into the first input of an **AutoGlyph** (from **Annotation**). Collect the output of the **RubberSheet** and the **AutoGlyph** before sending them to **Image** (Figure 12.3). Execute once.

Figure 12.3
Current VPE



It may be extremely hard to see, but DX places a small dot on the map with the same X, Y position as the probe.

- Open the CDB of the **AutoGlyph** and change the *type* to "text". Execute once.

Experiment with moving the probe point around in the scene. You can set DX to Execute On Change, so that as you move the probe the text is updated. Think about what the data values being displayed represent. Re-read the beginning of Chapter 3 if you don't remember.

OpenDX includes a second module that handles some of this work. Next, perform the same task as above, using **Pick** instead of **Probe**.

- Place a **Pick** (from **Special**) on the canvas. Wire its output up to the **Print**. Go into Pick mode using View Control. (Single) click somewhere on the surface and execute once.

Notice how the output of **Pick** is already a Field with a 'positions' component. There are some other components, but no 'data' component.

- Open the configuration dialog box for **Pick** and look at the *interpolate* input; set it to 1.

The interpolate option forces the output of **Pick** to find the object's data value at the position of the Pick.

- Copy the existing **AutoGlyph**. Wire the output of the **Pick** into its first input and wire the output of the **AutoGlyph** into the **Collect** (Figure 12.4).

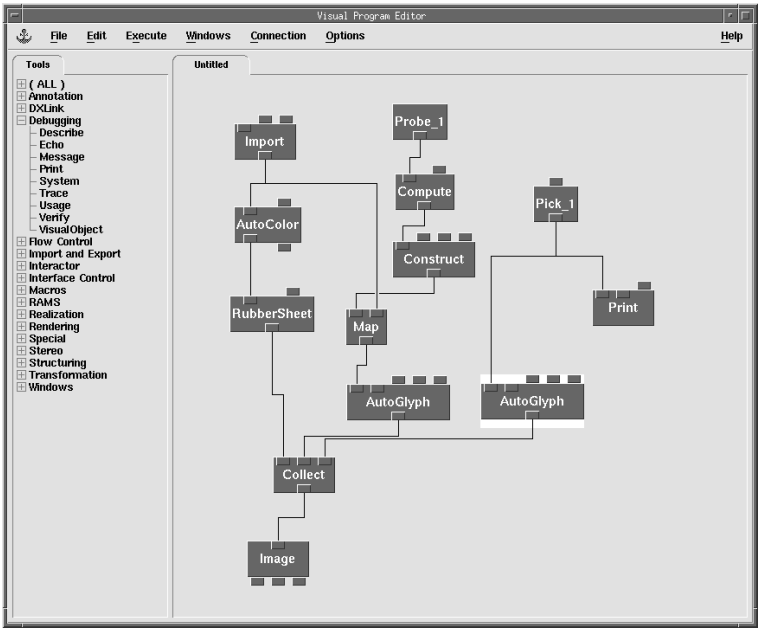


Figure 12.4
Current VPE

- Change DX to Execute on Change. Use the View Control to select the Pick mode. Try single clicking at different points in the Image Window.

Notice how **Pick** updates its value after every selection. Try picking outside the image; you should find that Pick returns a Field with 0 components, so no glyph is produced.

- Save the program as "pick.net"

Review

In Exercise One, you used the **ForEachN** module to loop over a series of integers and the **ForEachMember** to loop over the members in a series. **ForEachMember** can also loop over the

items in a list. This feature allows for looping over all data elements in a field. You used the **Extract** module to extract the data component and pass this into the **ForEachMember** module.

You also used **Done** to create a loop all by itself. However, you should have learned to be careful of using interactors to control **Done**, because Interactors have effect only at startup. Remember that OpenDX is a client-server program. While the server is executing a program, changes in interactor values are not processed. Using 0 as an input to **Done** creates an infinite loop. Though it wasn't illustrated here, you could write your own user interface to hook up to DX using the DXLink libraries. For example, you could define a button that interrupts an infinite loop by calling the End Execution routine.

You used **GetLocal** and **SetLocal** to accumulate a series of data. In the Flow Control category, there is another pair of modules named **GetGlobal** and **SetGlobal**. These are similar to **GetLocal** and **SetLocal**. Recall that each time OpenDX executes a loop with **GetLocal** and **Setlocal**, the loop is reinitialized to the "initial value" input. In contrast, **GetGlobal** and **SetGlobal** are NOT reinitialized until explicitly done using the "reset" input. Thus, the "global" modules are useful if you want to maintain state over the course of multiple loops.

Everything at the level of the looping tools is assumed to be inside the loop. This includes modules on separate pages, so it is generally best to put a loop inside a macro because only the modules within the macro are executed during the loop

When working with a package such as OpenDX, speed and memory play a significant role. OpenDX caches results after computation within modules. The results from loops and macros are also cached.

The **Sequencer** and looping have a great deal in common. However, only one **Sequencer** can exist in a visual program, whereas multiple loops can exist. The developer should use the **Sequencer** when user interaction is required or desired. For more extensive information on looping, refer to the chapter entitled "Data Explorer Execution Model" in the User's Guide.

Picks and **Probes** provide an easy to use interface for selecting and viewing specific data and objects. The **Pick** module is often easier to use but provides a higher level of complexity. For example, if a pick intersects multiple objects in a scene, then the output returns information for each object. The Probe module is simpler than the **Pick** module, but its use in complex applications may require additional user programming.

The initial output of a **Pick** that has not yet been used in an Image window is a Field with 0 components. Most modules accept this and simply pass the Field through. However, the **Probe**'s initial value is a NULL object–when passed to most other modules this causes an error. In this case, you would have to program the network with an **Inquire** and **Route** to shut off the **Probe** until it is initialized. For more information, refer to the User's Reference Manual.

Tips, Tricks and Memory Usage 13

Introduction

This chapter provides tips for working with large visual programs, rendering images and minimizing memory usage. Recall that Chapter 4 presented techniques for organizing large visual programs, including using pages and annotation. This chapter describes related techniques used to locate a specific tool in a large program and prevent overlapping modules from displacing adjacent modules on the VPE canvas. It also describes ways to prevent line aliasing, display two objects that occupy the same position in space, and troubleshoot some common image rendering problems. Finally, it describes the OpenDX execution model and its use of memory in some detail. Because OpenDX does not always use memory efficiently, visual programs hosted on computers with limited memory (<64 MB) or dealing with large data sets can run out of memory prior to completing the desired visualization. Awareness of how OpenDX uses memory can help the user reduce memory requirements and minimize these limitations.

VPE Tips

Module Overlap

As visual programs increase in size, the programmer is faced with the growing challenge of organizing the program and keeping track of different program elements. When a network becomes sufficiently large, it needs to be divided into separate pages, where each page ideally provides separate and distinct functionality. Logically, adding or moving modules around can cause some modules to overlap, and by default, modules are not allowed to overlap on the VPE canvas. Thus, when a module is placed too close to an existing module, OpenDX automatically displaces one or more modules and connections to avoid the overlap. There are two ways to modify the default automatic displacement feature (commonly known as bumper cars). The first option changes the displacement behavior, so that when a module is placed too close to an existing module, OpenDX moves the new or moved module into the nearest open space and does not displace the existing module. This option is engaged in the Visual Program Editor by selecting "Prevent Overlap" from the "Options" menu. The second option turns off displacement altogether, allowing OpenDX to permit overlapping on the VPE canvas.

Thus, modules may rest on top of each other. Some modules and connections can be obscured, but the result (particularly for experienced users) may be preferable in large programs. Enabling overlapping is more complicated than simply setting an OpenDX option, because characteristics of the underlying X-windows system for this user-login must be changed. To enable overlapping, the user must edit the ".Xdefaults" file in his/her home directory to add the line:

DX*vpeCanvas.allowOverlap: True

Every X session started after this file is changed recognizes the change and permits VPE canvas overlapping. The change can be incorporated in the session current during editing by executing the command "xrdb -merge .Xdefaults". The user who wants to use non-overlapping as the default, but be able to occasionally enable overlapping immediately should put the above line by itself in a file, e.g., named "olap", then use the "xrdb –merge olap" command when desired.

Layout Graph

When visual programs are built with modules placed haphazardly on the canvas, a network graph can become difficult to read and understand. Under the Edit menu there is a "Layout Graph" option–selection of this option results in the visual program automatically being reorganized so that modules are placed in a downward flow fashion. Often this is very helpful, but sometimes it can produce a layout that is even harder to read and understand than the original. Layout Graph can always be immediately undone using the Undo option in the Edit menu, so the user can always try it to see how well it works.

Finding a Module

As visual programs grow it becomes increasingly difficult to locate a specific module or tool. The "Find Tool" facility, found in the Edit menu, displays a list of currently used tools. Select the tool from the displayed list to be located then select the Find button. The VPE will highlight the first occurrence of the tool on the canvas. Click the Find button again to locate the next occurrence of the module. Receivers and transmitters can be selected in a similar manner by typing in their names. To determine which interactor goes with a particular stand-in, just double-click on the stand-in–OpenDX then highlights the corresponding interactor in the Control Panel. Or, select the stand-in and use "Show Selected Interactor" in Control Panel's Edit menu. To determine which stand-in goes with a particular interactor, select the interactor and choose "Show Selected Tool" in Control Panel Edit menu.

To keep better track of a large visual program or provide hardcopy documentation, you can print your visual program. To do this, use the "Print Program..." option under the File menu. Each visual program page will be printed on a separate page. There is an option to display the values of module inputs that are set as annotation on the printout. You can print the visual program as a postscript file or directly to a printer.

Image Rendering Features

Objects Sharing the Same Physical Space

If two objects exist in the same exact space, the rendering software will have a hard time determining which object to display. The "fuzz" attribute is used to solve this problem. Pass the object through the **Options** module, setting the value of the "fuzz" attribute to an integer to determine display priority, where higher values are displayed on top of lower values in rendering from a positive view direction. For example, if three objects are in the same space and have fuzz values of zero, one, and two, and the view direction is positive, the object with "fuzz" of 2 renders on top of the objects with "fuzz" 0 and 1.

Empty or Inappropriate Display

Sometimes a visual program will produce an Image window whose contents are not what are expected. For example, the Image window might be blank. Usually this results from use of inappropriate, previously defined camera settings. Use the Reset under the Options menu or in the View Control dialog of the Image window as a way to quickly reset the camera settings to point at the center of an object scene and render all existing objects. However, if two objects exist in the scene but the distance between the two is great, then a blank image can still result. The **ShowBox** module, which creates the outline of a box that encapsulates the objects in the display, can be used to determine if any objects are present. Simply pass the target output through the **ShowBox** module before sending the output to an **Image** or **Collect**. The result will indicate whether elements are present but not displayed in the current view, or no elements are present.

Another common problem is to zoom in too much or incorrectly rotate an object. To return to the previous image, there is an Undo button under both the Options menu and in the View Control window. Undo can cycle back through the previous 10 images. An accompanying Redo option allows the user to move forward through the same sequence of images, following one or more Undo operations.

Finally, during development an Image window can easily become buried under the VPE and other windows. Rather than move the VPE around or close windows, simply double-click on the Image tool in the VPE and the Image window will be moved to the "front" of the display. If the image is created using a Display module, select the Display module in the VPE and select "Open Selected Image Window(s)" from the Windows menu.

Line Aliasing

A common problem in rendering lines is that under-sampling the points along the line results in so-called aliasing, which makes the line thickness appear to vary or have missing sections. Anti-aliasing makes the lines appear smoother and more uniform by adding subtly shaded pixels, as illustrated in Figure 13.1. If your computer system provides hardware support for OpenGL 3-D rendering and OpenDX was built properly for your system, the software will support anti-aliased and multiple pixel width lines with hardware rendering. To have OpenDX render anti-aliased lines, simply pass the

renderable object through the **Options** module, and set the value of the "antialias" attribute to "lines". To specify multiple pixel width lines, pass the object through **Options**, setting the "line width" attribute to the desired number of pixels. If OpenDX is running on a system without OpenGL or GL installed, the anti-aliasing effect can be achieved by rendering the image at a larger size than needed, then using OpenDX facilities (e.g., the **Reduce** modules) or post-production software (e.g., Adobe Photoshop or ImageMagick) to reduce image size to that desired.

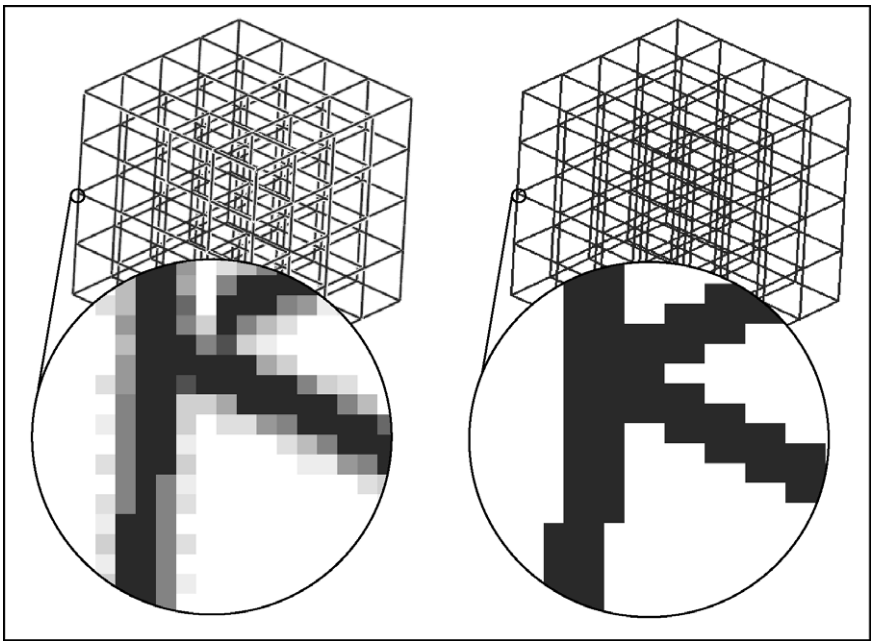


Figure 13.1
Anti-aliased and aliased images

Memory Usage

OpenDX is a memory intensive application, and thus prone to "Out of Memory" errors during execution on hosts with small memory or in applications involving large data sets. Several techniques can be used to reduce the amount of memory OpenDX uses to avoid memory problems. Understanding these techniques first requires understanding the OpenDX execution model and its use of memory.

The OpenDX *Executive* manages the execution of the visual program. The Executive processes the inputs and analyzes the program's flow to determine an efficient execution order for the different modules. Its default behavior is to run as a single process, passing objects "by reference" between modules. However, OpenDX provides two other execution behaviors, symmetric multiprocessor

(SMP) execution and distributed execution. Initial designs of the software were written with SMP hardware in mind, so the Executive was designed to allow it to segment data and tasks to run using shared memory and multiple processors. The default behavior of OpenDX on a multiprocessor machine is to start by using two processors; if there are more than four processors available, then OpenDX uses half the total number of processors. Alternatively, the user can specify the number of processors to use with the "–processors" parameter. The distributed execution model allows OpenDX to take advantage of a network of single processor machines. The user within the VPE can assign control of a module or a set of modules to one or more different machines and the Executive itself controls the details of distributed execution. Thus, facilities for parallel/distributed execution of visual programs are an integral part of OpenDX, when running in a multiprocessor or networked environment.

OpenDX Object Cache

During program execution, the Executive saves (i.e., caches) intermediate results for use if the program runs again or if the **Sequencer** is used to generate a sequence of images. By default, the Executive manages the cache, determining which intermediate results to store, which to remove as the cache fills during execution, etc. By default, the size of the OpenDX cache is set to a very large percentage of the physical memory available on the host. Thus, removing results from the cache is critical, since this is the primary way to reclaim memory needed to store new results. The size of the cache can be explicitly controlled using the "-memory" option when starting OpenDX. Minimally, cache size must be at least as large as the maximum amount of memory required by any module in the program. For most programs, increasing the cache size generally speeds up execution, but the cache size cannot exceed the amount of virtual memory available to OpenDX.

Default Memory Size and Paging Space

OpenDX's default memory allocation scheme is initially determined by the amount of physical memory available to the system. Unless noted in implementation-specific documentation, if there is less than 64 megabytes of physical memory, memory allocation will use all but 8 megabytes of the physical memory; if there is more than 64 megabytes of physical memory, OpenDX will grow to use 7/8 of the amount of physical memory. At present, physical memory is currently limited to two gigabytes. The default amount of memory that can be used can be set explicitly using the "-memory" parameter when starting OpenDX, or the "Memory" field of the Connect-to-Server Options dialog. Since it is possible for OpenDX to use a large amount of virtual memory, the host workstation should be configured with paging space at least two or three times the total physical memory in the workstation. If the host has insufficient paging space, the host operating system may kill OpenDX or other processes, often without warning. Thus, particularly for use with large, complex visualizations, OpenDX, physical memory, and paging space need to be appropriately matched by the system administrator when the system is initially installed.

Reducing Memory Requirements

Besides using the "-memory" option, there are several other methods to reduce the memory requirements if OpenDX still does not have enough memory to execute a visual program. Some memory saving techniques include not rendering image data, using delayed and/or byte colors, using "speedy" glyphs, converting to different data types, importing one series member at a time, and reducing the grid resolution.

A common mistake by developers is to render image data (i.e. 2-d arrays of scalar values) using **Render**, **Image**, or **Display** with a camera input. This results in interpreting the image as a very large number of quads. Instead, the image can be passed directly to **Display**, without a camera input. If the field does not contain a 'colors' component before the **Display**, a module such as **AutoColor** can add one. By default OpenDX uses three 32-bit floating point values (96 bits) to describe each color value. This is much greater color resolution than is required for many applications. In such cases, the delayed colors option can be specified to use a single scalar byte as an index into a 256 value color table. To specify delayed colors, convert the data component to unsigned byte and set the 'delayed' input in the **Color** module to 1. Yet another option to reduce the amount of memory for colors is to use 3-byte color descriptors. Set the DXPIXELTYPE environment variable to "DXByte" or start OpenDX using "dx -optimize memory" to force colors to be 24-bit instead of the default 96-bit color descriptors. Note that this setting affects both **ReadImage** and **Render** tools.

If the visual program requires use of glyphs (**AutoGlyph** or **Glyph**), the programmer can select less ornate, less complex glyph objects to reduce memory consumption. These "speedy" glyphs have fewer connections and therefore consume less memory. Set the **AutoGlyph** or **Glyph** 'type' parameter to either "speedy" or to a small fraction of 1.

In many situations, it may be acceptable to convert data components to types that require less space by using the **Compute** module. For example, floating-point data can be changed to byte data, using an expression in **Compute** such as: byte(255*(data-min)/(max-min)). Note that such a reduction reduces the space requirements of ALL downstream modules, and can thus result in a dramatic decrease in total space used.

When working with series data, an easy way to reduce the memory requirements is to import one series member or slice at a time instead of the whole series. This reduces the memory requirements by not having the whole series in memory at once.

Finally, if it is possible to sacrifice resolution in the data set, the **Reduce** module (usually right after **Import**) can be used to reduce the number of points in the data set. The **Reduce** module requires regular connections, so this technique is applicable only if the data set is on a regular grid. Again, the earlier such a reduction is made the greater the cumulative effect.

Cache Control: Executive

As noted above, OpenDX stores intermediate results in its object cache, and to some extent the cache can be explicitly controlled by the user. For example, the Import module reads in a data set and caches it within OpenDX. If the data file is changed and the program simply re-executed, the program continues to use the old data. That is, OpenDX must be forced to reread the input file, because the Executive's default behavior is to use the results cached from the previous execution. The Reset Server option in the Connection menu flushes the cache, and also forces OpenDX to (re)read the (updated) data file.

The user can control how the Executive caches intermediate output values from each module. For any specific module, simply open the module's Configuration Dialog Box and look at the Cache option menu to the right of each output. In general, it is most efficient to cache only the output of the last module in a "straight line execution" sequence. That is, unless unusually difficult computation is involved in this sequence, it is better to reduce memory in each computation than to try to decrease computation in the "next" program execution. It is important to note that the cache of the Import module refers to its result, not its input. Even if the Import module cache is turned off, the default behavior of OpenDX is to cache the result obtained by actually reading an external file. This is important if the input file is being changed while repeatedly executing the visual program.

Caching can be turned off altogether using the "-cache off" execution option. However, the effect on execution can be dramatic, so this is not generally a good idea. A better alternative, at least as a starting point to improve memory utilization, is to use an option in the Edit menu of the VPE called Output Cachability -> Optimize. This causes OpenDX to use a heuristic to try to optimize the caching behavior of each tool in the program.

Display and Image Cache Control

Some modules use a special internal caching system to cache local data other than final results to simplify recalculation if the program is re-executed with the same parameter values. The most important internal caches are maintained by the **Display** and **Image** modules. Most of the time this is desirable because it speeds up execution, but in some cases it is preferable to turn off the **Display** and **Image** caching. For example, turning off the internal image cache can be useful if the program is running as a batch job that generates images for external storage only. To turn off the internal caching, open the Configuration Dialog box for the Image module and select "No Results" in the column labeled "Internal Caching" (Figure 13.2). For the **Display** module, the value of attribute "cache" must also be set to "0" using an **Options** module. Note that the "-cache off" execution option mentioned above has no effect on OpenDX internal caching.

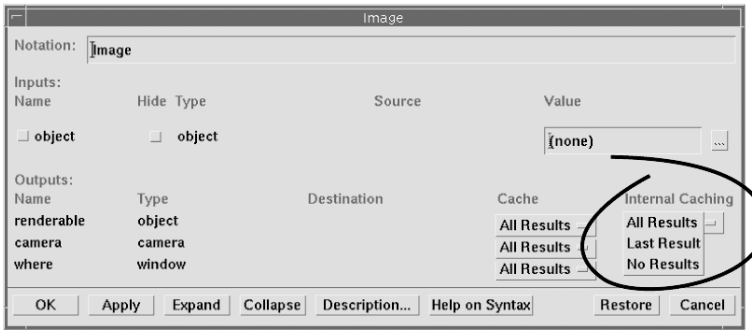


Figure 13.2
Image Internal Caching

Per Process Limits

As a final note, some systems may enforce per process limits on such things as data segment size, stack size and so forth. A user who encounters such limits should work with the host system administrator to adjust these parameters to run OpenDX most effectively and efficiently, particularly when using large data sets.

Conclusion

Managing the features of the VPE can increase the developer's productivity. Understanding the advanced features available for rendering can increase the speed on systems with hardware assistance. Understanding and managing aspects of the OpenDX Executive can help balance processing time and memory utilization, a balance that is critical in the manipulation and analysis of large data sets.

Matching the configuration of the host system with intended use in running OpenDX can also help produce a more effective and efficient visualization environment. Inattention to this sort of execution detail often leads novice users to abandon their goals at the first "Out of Memory" error, whereas attention to this detail allows the more experienced user to perform much more complex tasks on the same host.

Camera Animation and Arranging Images 14

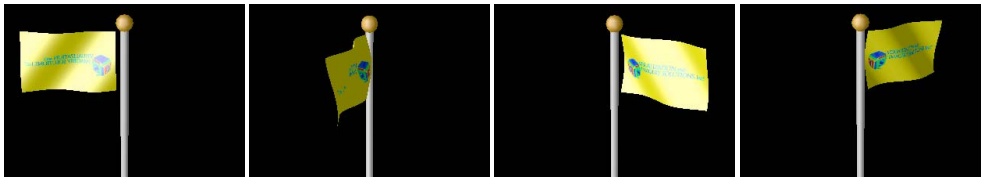
Rationale

A majority of data visualization applications focus on showing scientific information through printed imagery. In order to show all aspects of the data, it may be necessary to show multiple view angles, time segments, or data subsets. Within OpenDX, you can easily combine multiple rendered images together into a single conglomerate using prewritten modules such as **Arrange**. However, the visualization may be more effective if produced as an animation that shows different view angles. Using the sequencer as a time step generator, you can move the camera position around in a scene and generate a sequence of images with different viewpoints. This chapter will show techniques for creating conglomerate images as well as creating sequences by moving the camera position.

Exercise 1. Camera Animation

Capturing images as a variable changes over time is a way to create animation. The changes in variable value may come from a time-series data set, result from moving the viewpoint of the camera, be produced by changes in a module's input value (e.g., **Isosurface** value) or even result from geometry changes (e.g., different slab positions). In earlier chapters, you produced simple animations by two of these techniques: using time-series data and changing slab positions (while keeping the camera position constant). The technique of moving the viewpoint of the camera around in a static field or in a dynamic field with one or more variables changing can often provide additional insights into the data. As an example, you will expand the waving flag exercise so that the camera's position changes over time. More specifically, the goal is to add a flagpole to the waving flag of logo.net from Chapter 8, then move the camera in a circle around the flag. The final result should be a simple sequence of images, similar to those in Figure 14.1.

Figure 14.1 *Animation Sequence*

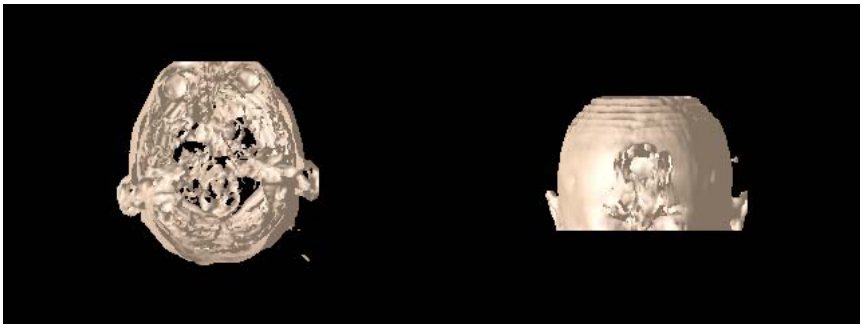


Exercise 2. Arranging Images

As visualizations increase in complexity, it is often desirable to provide multiple views of the same object in the same image, rather than capturing one view per image. This provides the viewer with more information and a greater opportunity to visually compare the multiple views and understand the information better. It is always possible to render multiple views separately, then combine them into a single image in a postproduction process with tools such as Adobe® Illustrator®. However, OpenDX has the facilities that make it relatively easy to combine and arrange multiple components into a single image, thus avoiding the need for post-processing.

The goal in this exercise is to import the mrb.binary data set and produce a single image with two views of an Isosurface of the MR scan, as shown in Figure 14.2. The Supervise functionality of OpenDX then allows you to rotate or pan each isosurface separately within the single composite image.

Figure 14.2
Arranged Images



Step-by-step instructions for Exercises

Instructions for Exercise 1

Start by opening the "logo.net" file you saved from Chapter 8.

To add a flagpole you will use three modules: **Construct**, **Tube**, and **ShowConnections**.

- Turn on the AutoAxes or use the **Print** module to determine the flag's positions. From those, determine the endpoints of a line that will act as a flagpole, i.e., that traces the edge of the flag that will be attached to the flagpole and extends downward to give the pole an appropriate height. Use a **Construct** module to create this line. The values for **Construct** should be something like the following: *origin* {[-50, -2000, 0]}; *deltas* {[0 2370 0]}; *counts* [1 2 1]; and *data* {5}. Now, use **ShowConnections** to display the line. **Collect** it and the flag, then send the combined result to **Image**.
- At this point the pole consists of only a line. To form a more realistic pole, **Tube** the line and change its **Color** to "grey75". The resulting visual program is shown in Figure 14.3.

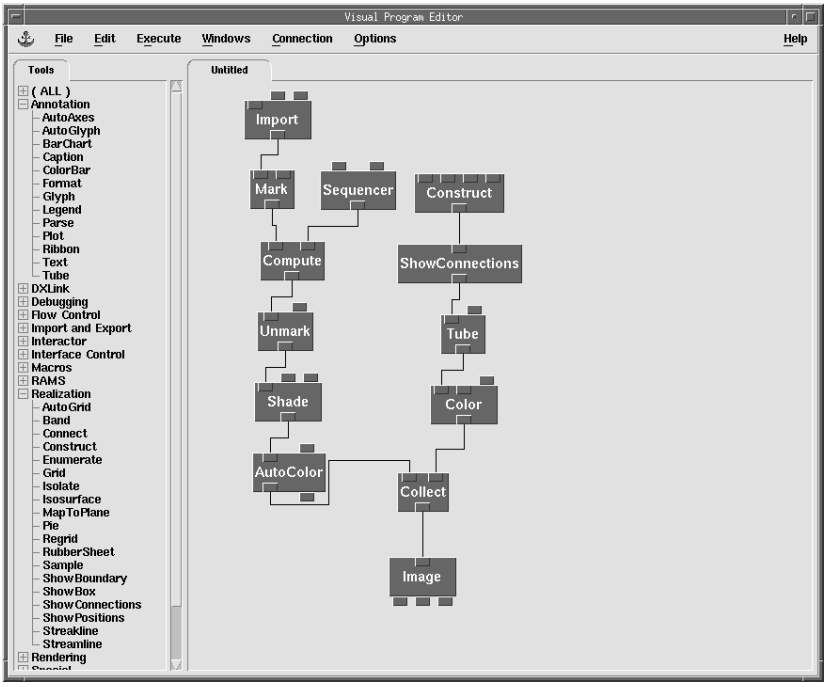


Figure 14.3
Current VPE

- Create an even more realistic flagpole by using **Construct**, **AutoGlyph** and **Color** to construct a gold ball on top of the flagpole. You center the ball at the top of the pole using values for **Construct** something like: *origin* {[-50, 400, 0]}; *deltas* {[0 0 0]}; *counts* [1 1 1]; and *data* {1}. Set the *scale* of **AutoGlyph** to 50 and color it "goldenrod", giving the program in Figure 14.4.

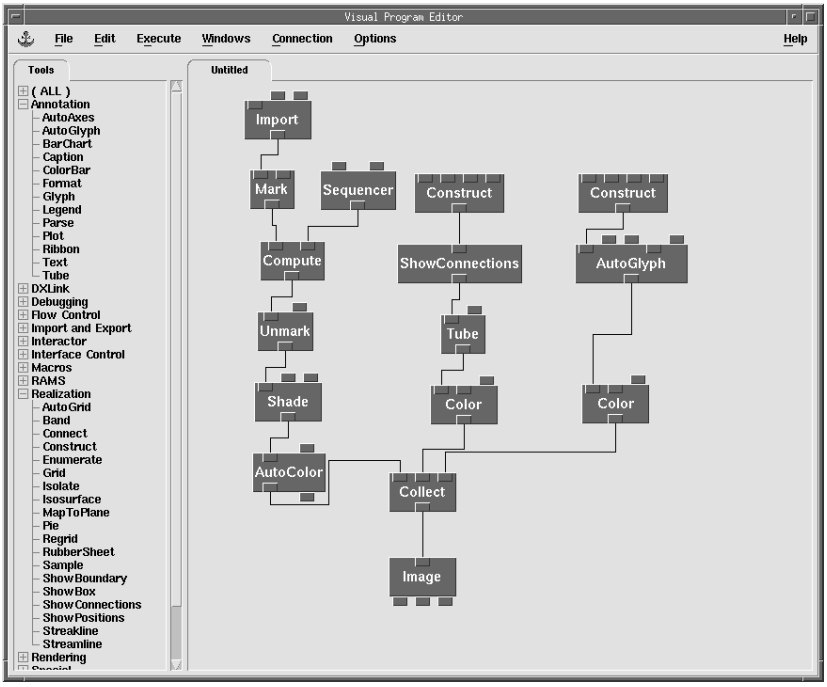


Figure 14.4
Current VPE

In order to move the camera around in a 3-D space you need to have a starting point from which to base camera movement, along with a plan which describes the additional points at which you will position the camera to capture other images. For example, given a starting point you can use polar coordinate translations to easily calculate new positions for the camera moving in a circular fashion in the same horizontal plane around the target object.

- Within the Image window, use rotation and zooming, etc. to choose a nice starting position for your animation.
- Place an **UpdateCamera** module (from **Rendering**) on the canvas near the **Image** module and examine the inputs. Wire the **Image**'s camera to the first input of **UpdateCamera**.
- Place a **Display** module on the canvas. Connect the output of the **UpdatedCamera** and the "renderable" output of **Image** to Display, as shown in Figure 14.5.
- Connect the "camera from" output of **Image** to an **Inquire** module, and use this to determine the origin in the calculation of the new camera positions.
- Place a **Compute** on the canvas between the **Inquire** and **UpdateCamera** modules and wire the **Sequencer** into the second input of **Compute,** as shown in Figure 14.6.

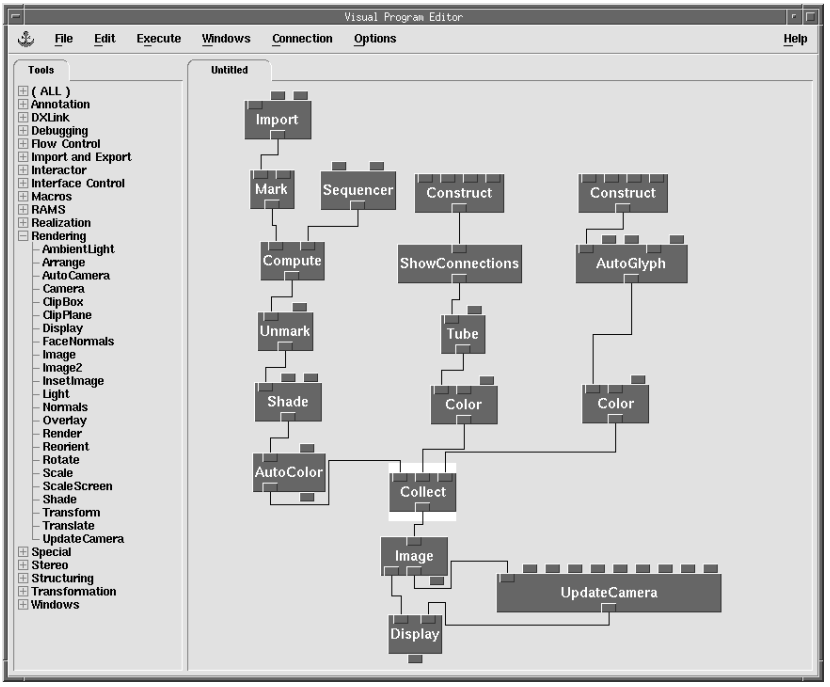


Figure 14.5
Current VPE

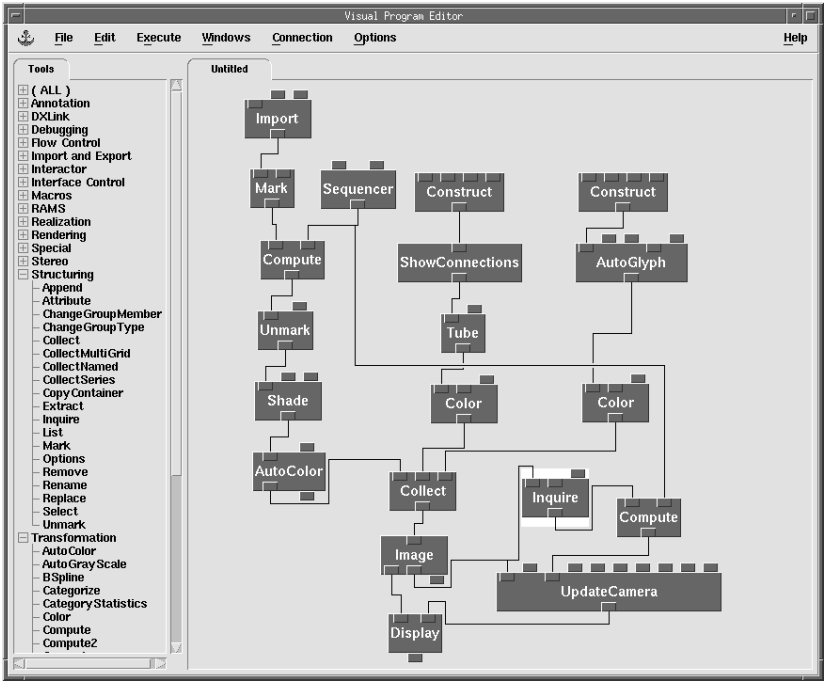


Figure 14.6
Current VPE

Notice that you can't use a second **Sequencer** here, since only one **Sequencer** is allowed per program. However, you can produce different values from an existing **Sequencer** by connecting its output to a **Compute** that transform the input sequence values to produce the new sequence needed.

- Use a multi-line compute statement to calculate the positions of the camera in a circular position around the flag. Enter the following as the expression for the **Compute**, entered all on one line with parts separated by semi-colons: theta=atan2(a.x, a.z); r=a.x/(cos(theta)); x=r*cos(b*6.28/10.0); z=r*sin(b*6.28/10.0); y=a.y; [x,y,z]
- Execute the program using the **Sequencer**.

This illustrates that functions within **Compute** modules can be quite complex. You can define multiple computations by separating them with semicolons. You can also assign values to temporary variables to simplify expressions. By defining multiple camera points you move the camera around the waving flag, capturing a new image at each camera point. You can make the motion smoother by increasing the number of camera points and images rendered. For example, increase the number of sequencer steps to 50, and in the expression above, divide by 50 instead of 10.

- To get the camera centered on the flagpole, set the *to* point of the camera to [-50 0 0] in the **UpdateCamera** and set the *up* vector to [0 1 0].

Instructions for Exercise 2

- Start a new visual program.
- Import the variable "pd" of "mrb.binary". Remember that you created a general file import for this data set in an earlier exercise.
- Create an isosurface, color it "peachpuff" and feed it to an Image.
- Place another **Image** tool on the canvas and feed the colored isosurface to it as well.

Notice how the images are independent of one another, i.e. their viewpoints, sizes, and rendering options can be different. However, the goal of this exercise is to place both images in the same Image window, so you can't simply display the two views in two different Image windows. What you need to do instead is to put together modules that perform the same rendering function as Image, but produce a rendered object as a result instead of producing a complete Image window.

- Replace each of the **Image** modules with a **Render**, **AutoCamera**, and **Display** combination (from **Rendering**), as shown Figure 14.7.

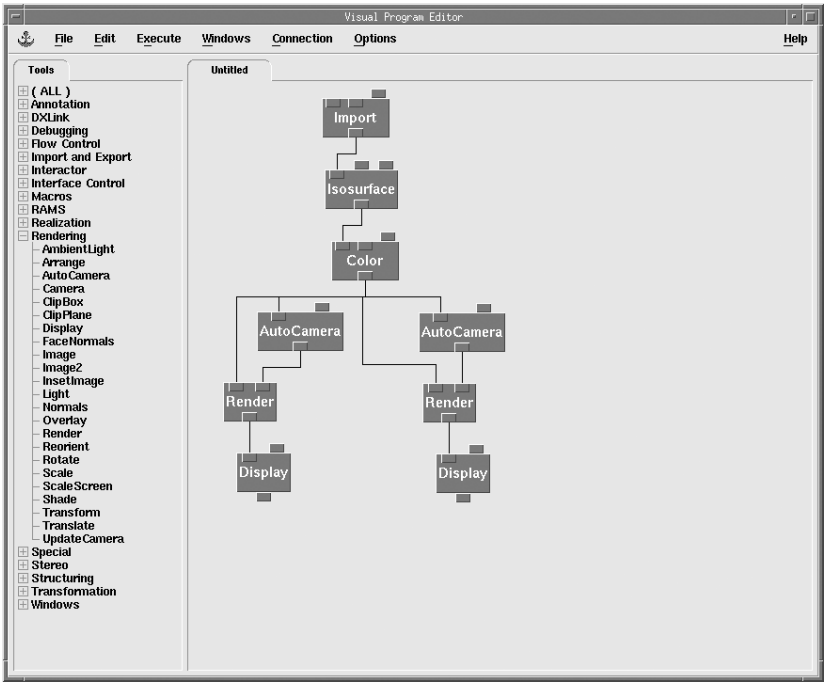


Figure 14.7
Current VPE

The new program still produces two images of the isosurface, but the viewing functionality associated with **Display** is not as varied as with **Image**. Notice that you cannot rotate or zoom on the Display windows as you could with the Image windows. In general, the **Image** module provides interactive display control that **Render** and **Display** do not. To control the "viewpoint" of the **Display**, you must program your network to explicitly change the inputs of **AutoCamera**.

- Change the input parameters so that both images are smaller and the viewpoints are different. Change the *direction* parameter on one of the **AutoCamera**s to “top”. Select the Expand button in each of the **AutoCamera** CDBs, then change the *resolution* parameter to 320.
- To combine the two images, add a **Collect** module and pass the outputs of the two **Render** modules to it. Pass the output of **Collect** to **Arrange** (from **Rendering**) and pass **Arrange** to one of the **Display** modules. Delete the other **Display** module, giving the program shown in Figure 14.8. Execute.

You can combine together as many images as needed by adding more inputs to **Collect**. You can arrange the images in various ways using the parameters of the **Arrange** module.

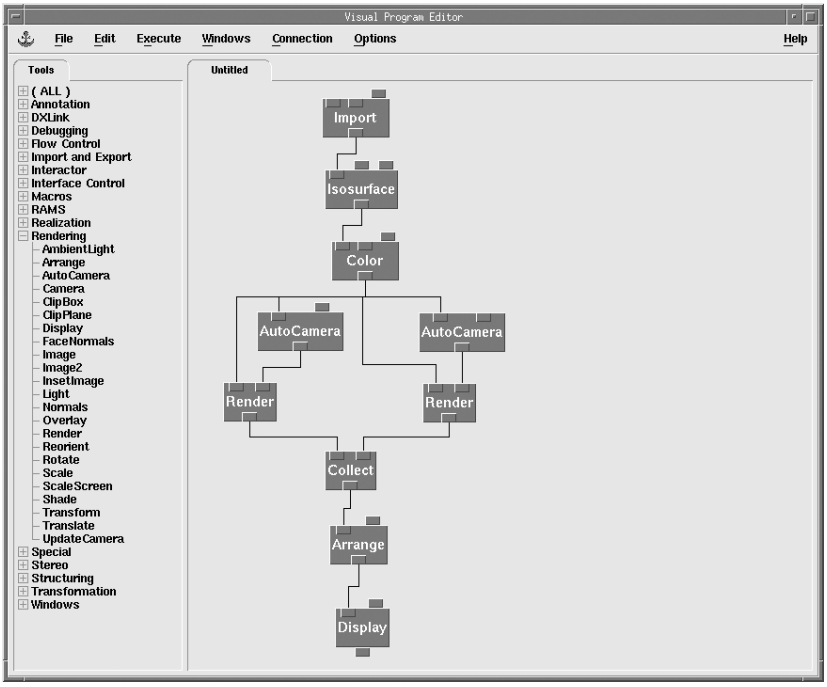


Figure 14.8
Current VPE

It is possible to add user interactivity to the arranged images using Supervise modules. The Supervise modules are more complex to use than **Image** or **Display**, but they give you a high degree of control over window behavior. Several macros already exist that can be used to provide the higher level of control, yet hide some of the complexity.

- Select Load Macro... (from the File menu) to load the macro ArrangeMemberMacro.net from OpenDX's sample macros directory (typically in /usr/local/dx/samples/macros). The new macro will appear in the Windows category.
- Place two copies of **ArrangeMember** and a single **SuperviseWindow** (from **Windows**) on the canvas.

SuperviseWindow creates and manages windows, while **SuperviseState** acts on events that occur within a window, such as mouse movement or click events. The **ArrangeMember** macro internally includes its own **SuperviseWindow** and **SuperviseState** modules, along with a display to render the object. With a combination of these modules, OpenDX associates a "parent" window with the **SuperviseWindow** module, and places "child" windows associated with **ArrangeMember** macros within the parent window. **ArrangeMember** will create, manage, and control the events of the child windows. Whereas **Image** provides only the default interpretation of mouse actions in its window, a Supervise module allows the programmer direct control over the interpretation of mouse events within a window. The developer can use this capability to define a much wider range of customized event interaction modes, to suit the particular application. However, due to the complexity of event

models and event programming, this topic is beyond the scope of this exercise. Only the predefined interaction modes are used here.

The first output of **SuperviseWindow** is the "where" parameter of its window, which simply identifies the window. Since this window is to be the parent of the two windows created by the two **ArrangeMember** macros, the modules must be wired together.

- Wire the *where* output of the **SuperviseWindow** module to the *parent* input of both **ArrangeMember** macros. Remove the **Display** module from the canvas since the new modules will render the image. The network should now resemble Figure 14.9.

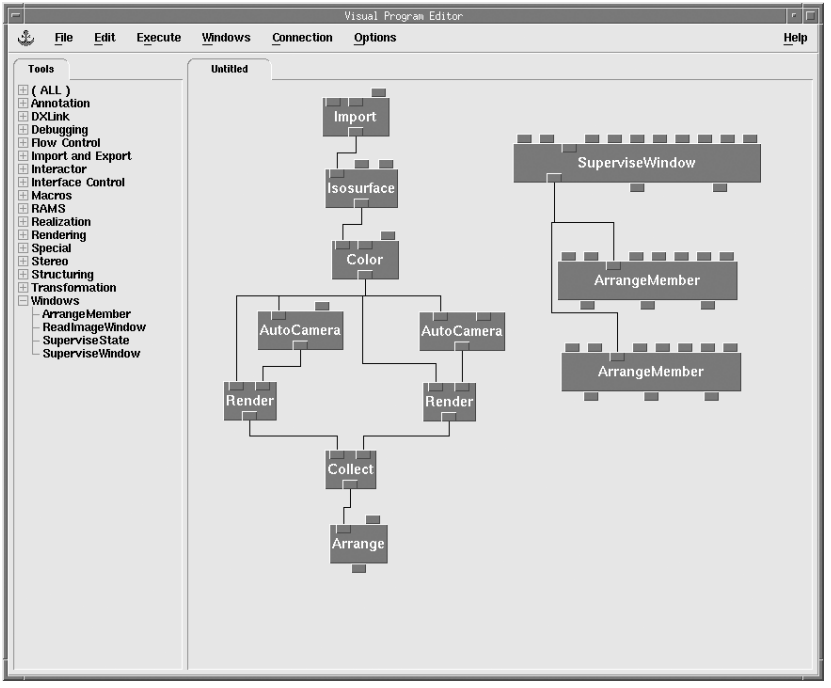


Figure 14.9
Current VPE

The **ArrangeMember** windows need to know the size of the parent window in order to render at the appropriate size. One of the outputs from **SuperviseWindow** provides this information and can be plugged into the two macros. Since the **ArrangeMember** macros must also know what objects to render, you must pass the isosurface to each macro.

- Wire the *size* output of **SuperviseWindow** to the *parentSize* input of both **ArrangeMember** macros.
- Wire the **Color** output to the *object* input of both **ArrangeMember** macros, giving the program in Figure 14.10.

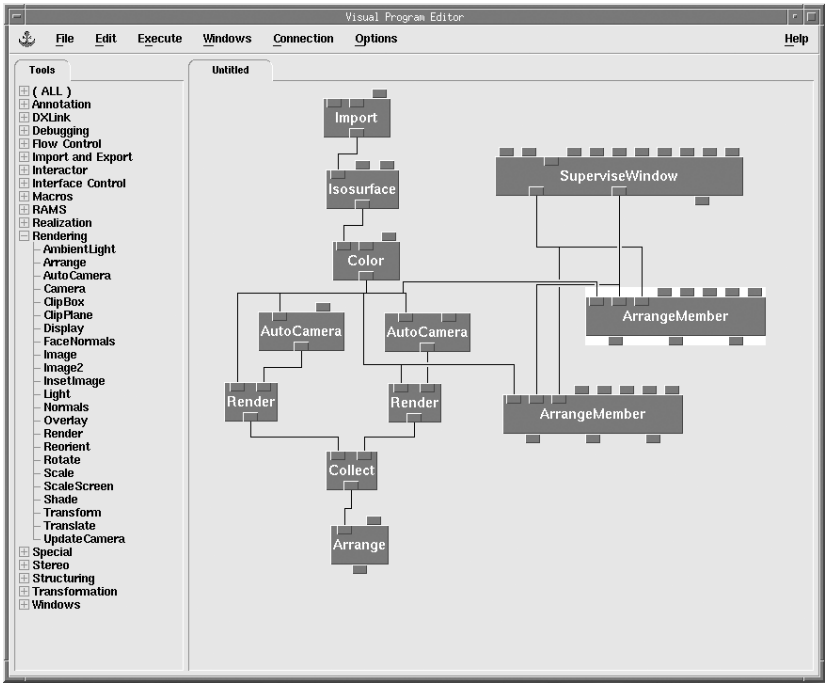


Figure 14.10
Current VPE

Although the wiring is complete, the **ArrangeMember** macros need more information before the network can execute. Parameters such as *totalSubimages*, *nHorizontal*, and *which* must be specified. *totalSubimages* is the number of images that are going to be pasted together. *nHorizontal* is equivalent to the horizontal input for **Arrange**. *which* identifies the pane in which the image is to be placed (this index always starts at 0).

- Open up the configuration dialog box for both of the **ArrangeMembers**. Set "*totalSubimages*" to 2 and "*nHorizontal*" to 2. Set "*which*" to 0 on one of the macros and to 1 on the other. Execute once.

OpenDX should display an "arranged" image with two isosurfaces.

- Set the *interactionMode* to 0 (for rotation) for each of the **ArrangeMember** macros. Select Execute on Change from the Execute menu and try clicking and dragging on the images.
- Try setting the *interactionMode* for one of the **ArrangeMember** tools to 1 (for pan).

Notice that the **SuperviseState** pan works differently than the pan mode in the **Image** tool. This illustrates that functional interactions can be implemented completely differently in different types of display tools.

Review

There are many ways to create animation. The first exercise shows how camera movement can be used to produce animation. Note that simply rotating the object, instead of moving the camera, could be used to produce the same animation. However, much more sophisticated camera paths can be produced using modules such as **BSpline** or **PathInterpolation** (a custom VIS, Inc. module), creating a smooth series of camera points that is difficult to duplicate by object rotation.

As illustrated in several earlier exercises, the **Image** tool provides an easy way to render and display objects, while also providing the user interactive control over the viewpoint, size, etc. While the **Image** tool is very easy to use, it does have disadvantages, notably that only a default set of behaviors is supported. If the users don't like the way **Image**'s interaction works they must look for alternative display mechanisms. For example, a user might want to define "zoom" as provided by a zoom rectangle starting at one corner and ending at the other, which involves the more specialized programming illustrated in the second exercise. This type of programming is also required if OpenDX is being used to produce imagery for a custom application that runs outside the normal OpenDX user interface. In such a case event control must be written to correspond to the application's "look and feel", rather than OpenDX's default behavior as defined by Image.

Interactive arrangement of component images is not possible using **Image** or **Display**. To provide interactivity, each individual panel must be a separate interactive image, which is then brought together to be arranged. The **Render/Display** method shown in this exercise provides a method to explicitly control the viewpoint using the **Camera** tools, but these modules don't provide direct interactive arrangement.

The Supervise modules provide another way to create images. While more complicated to use, they give complete control over windows to the programmer. Interaction modes, such as mouse clicks or keyboard presses, can be programmed to affect the object or camera in any way. The example in /usr/local/dx/samples/supervise/complexdemo illustrates how this can be used to move one object in a collection of objects independently of other objects in the collection, and how to create a "point and type" caption which can then be dragged around using the mouse.

For authors creating applications on top of OpenDX, the Supervise modules provide irreplaceable functionality. Suppose the developer requires an application with its own graphical user interface. This interface can communicate with the Data Explorer modules in essentially two ways. Using the DXCallModule interface, the developer can individually call OpenDX modules much like subroutines. Thus, an application program can invoke **Import**, **Isosurface**, the Supervise modules, and finally **Display**. In this mode, the program must directly control both the order of module execution and memory management for the system. Alternatively, the developer can use the DXLink Developer's toolkit, which allows an application to run OpenDX remotely, passing values into a visual program and receiving results back from the visual program. In this mode, OpenDX retains control of module execution and memory management, so that the developer can separate control and memory concerns of the application and the visual program.

Constructing a Native DX File 15

Introduction

Often data sets are provided in a format that cannot be easily imported into OpenDX via the Data Prompter. A detailed, real world example is described below to allow you to work through a typical data import problem. The example is based on the National Atmospheric Deposition Program to provide acid rain values on-line from their web site http://nadp.sws.uiuc.edu/. The data sets available from this Web site are stored in a manner designed to allow a user to easily select data from a particular data collection station. The time series information for each individual station is stored as an ASCII formatted table, with fixed width fields delineated by either commas or tab characters. The location of each station is available on each individual station's general description page. Importing data from a single station is easy using **ImportSpreadsheet**. However, importing data from multiple stations as time series data, with location information intact, is quite difficult without using the native OpenDX file format.

You should work through this example to gain experience in how to handle data that cannot be imported into OpenDX using the other available importing techniques. In effect, the process using the native file format proceeds along the lines of trial and error, where you construct the OpenDX native file by progressively adding information and testing the file along the way. If you have a significant programming background you are encouraged to think about the correlation between the program that created the file and the native file description that you are incrementally constructing. Programming tools such as Perl can be used to parse text files like these and write out an OpenDX native file format.

Description of the Data Files

There are five data files needed in this exercise, each downloaded from the National Atmospheric Deposition Program's Web site. The first, "notes-depo.html", is an HTML file describing what the variables represent within the other data files. The second, "station-id.txt", is the description of each station for which data is reported on the Web site, including important location information. These first two files are crucial, no matter what specific sites a user may choose. The last three files are

representative time series data sets, in this case the time series tables for the three collecting stations in the state of Nevada.

Because collection stations are positioned based on physical properties rather than by virtue of being at intersection points on some arbitrary grid, collection station data must be interpreted as scattered data. Thus, you cannot and will not create a 'connections' component to organize data from different stations. However, there is regularity in the data associated with each collection point, in the sense that each station's time series contains data for the same period of time, by years from 1985 to 1993.

There are two ways to structure this data into a native DX file. One way is to create separate fields for each station, then group the fields together at the end. The other way is to create one field that contains all of the stations' information. If you create a single field, you can use a Connect module to connect scattered data of the field; on the other hand, you cannot connect separate fields contained in a group. Therefore, your ability to easily manipulate the data will be enhanced if you construct a single field for this example.

Instructions for Exercise

- Begin by creating a new blank file named nv.dx. On the first line, place a few comments by starting the lines out with a # symbol, as shown in Figure 15.1.

Figure 15.1 *First Line*

```
# Nevada deposition amounts, time series for 1985-1993
#
# Native DX file
#
```

The first information that needs to be placed in the file is the positions of each station. Within the "station-id.txt" file, the longitude and latitudes are stored; however, we must convert them from degrees-minutes-seconds to decimal degrees.

- Create an object definition, then place the locations below it (complete syntax for creating an object can be found in IBM's DX User's Guide in Appendix B under the Native file format). Start with the keyword "object" and assign the object a unique number. Next construct an OpenDX description of the data and add the keywords "data follows". On the next three lines, place the three position pairs calculated from the longitude and latitudes. Remember that you can put comments wherever needed. Figure 15.2 illustrates the object definition, using comments to separate each line of the definition.

Figure 15.2 *Object 1*

```
# Nevada deposition amounts, time series for 1985-1993
object 1 class array type float rank 1 shape 2 items 3 data follows
#NV00
115.4255555 36.13583333
#NV03
119.2566666 38.79916666
#NV05
114.2158333 39.005
attribute "dep" string "positions"
```

Since you are working with time series data each time slice, or "series position", is a separate object that references the positions (object 1 above). The next step is to add data to the native file. Each variable from the data files must be defined as a separate object. For example, the first column of each of the data files is the "station-id". This is also the first data value for each time slice.

Create a new object to stores the station-id data. Make sure to add the attribute for dependency, as shown in Figure 15.3.

Figure 15.3 *Object 2*

```
#
# object 2 is row 1 column 1 of the spreadsheet. It is rank 1 because
# strings are stored as vectors and shape 5 because it needs to store
# the 4 char plus the end of line character. This is the same for all
# the rows and columns so we only need to construct it once.
#
object 2 class array type string rank 1 shape 5 items 3 data follows
"NV00"
"NV03"
"NV05"
attribute "dep" string "positions"
#
```

There is a lot of information stored for each collecting station that may not be required in a particular application. For this example, we include only the "CA" values, skipping the "percent completeness".

Create an object that contains the CA values for each position from 1985. Look at row 1 in each of the nv data files. The resulting description is shown in Figure 15.4.

Figure 15.4 *Object 3*

```
#
# CA values
object 3 class array type float rank 0 items 3 data follows
1.0
0.05
1.99
attribute "dep" string "positions"
```

Now you have enough information entered to construct an actual field object, which you need in order to test the data file.

- Create a new field object and attach the other objects as appropriate components. Note that the component values are the unique numbers assigned to the objects. The resulting definition is shown in Figure 15.5.

Figure 15.5 *Field Test Object*

```
#
# Field object
object "test" class field
component "positions" value 1
component "id" value 2
component "data" value 3
```

- Save the native DX file as “nv.dx”.
- Start the OpenDX VPE and construct a visual program with an **Import** set to import the native DX file you just created, connected through **AutoGlyph** then **Image**, as shown in Figure 15.6.

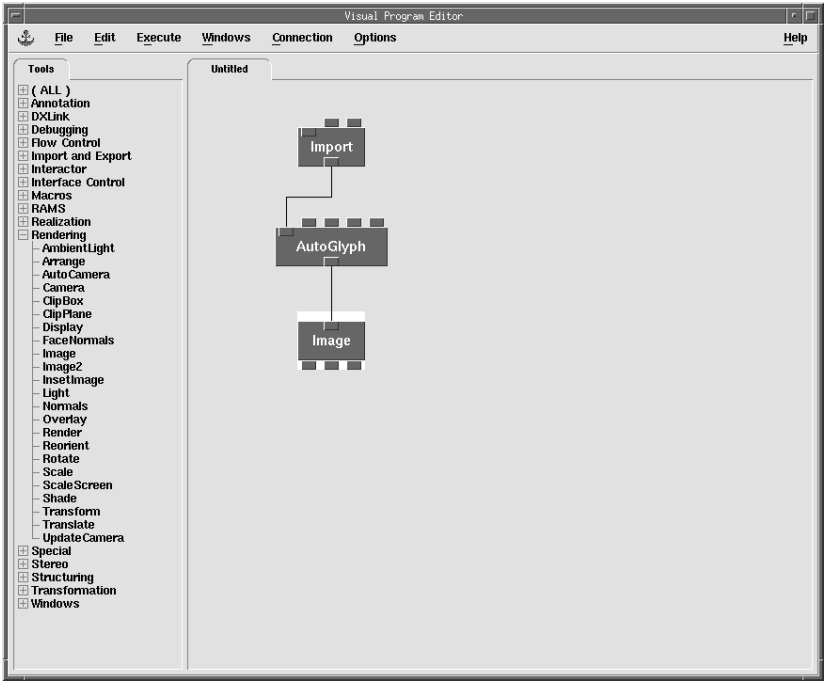


Figure 15.6
First Program

- Execute the program once. You should see three points that correspond to the CA values. If you get an error, make sure to check for typos in “nv.dx”.

The native file now describes a field with 'positions' as object 1, a component named 'id' as object 2, and a 'data' component as object 3. This description seems to be the right approach, but it is clearly incomplete. Continue to add more of the data values from 1985 as separate components.

- Add another object in "nv.dx" to define the Mg values after the CA values. Continue by defining similar objects for K, SO4, and % Ppt Rep. by F Chem, resulting in a definition similar to the one in Figure 15.7.

Figure 15.7 *Mg, K, SO4, and %Ppt added*

```
#
# Mg values
object 4 class array type float rank 0 items 3 data follows
0.142 0.011 0.234
attribute "dep" string "positions"
#
# K values
object 5 class array type float rank 0 items 3 data follows
0.054 0.012 0.143
attribute "dep" string "positions"
#
# SO4 values
object 6 class array type float rank 0 items 3 data follows
1.26 0.15 2.99
attribute "dep" string "positions"
#
# % Ppt Rep.by F Chem.
object 7 class array type float rank 0 items 3 data follows
0 0 0
attribute "dep" string "positions"
```

- Remove the previously constructed field object. Before entering the 1986 data, construct a field object that references all of the 1985 data and has a 'series position' attribute set to 1985. The entire data file should look similar to the one in Figure 15.8.

Figure 15.8 *1985 data field*

```
# Nevada deposition amounts, time series for 1984-1993
#
# Native DX file
#
object 1 class array type float rank 1 shape 2 items 3 data follows
#NV00
115.4255555 36.13583333
#NV03
119.2566666 38.79916666
#NV05
114.2158333 39.005
attribute "dep" string "positions"
#
# object 2 is row 1 column 1 of the spreadsheet. It is rank 1 because
# strings are stored as vectors and shape 5 because it needs to store
# the 4 char plus the end of line character. These are the same for all
# the rows and columns so we only need to construct it once.
```

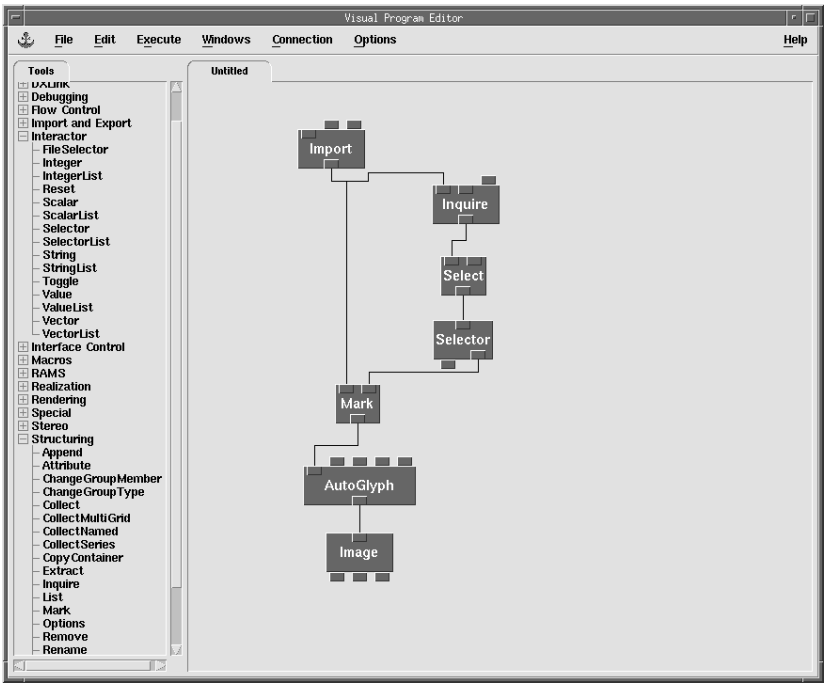
```
#
object 2 class array type string rank 1 shape 5 items 3 data follows
"NV00"
"NV03"
"NV05"
attribute "dep" string "positions"
#
# CA values
object 3 class array type float rank 0 items 3 data follows
1.0
0.05
1.99
attribute "dep" string "positions"
#
# Mg values
object 4 class array type float rank 0 items 3 data follows
0.142 0.011 0.234
attribute "dep" string "positions"
#
# K values
object 5 class array type float rank 0 items 3 data follows
0.054 0.012 0.143
attribute "dep" string "positions"
#
# SO4 values
object 6 class array type float rank 0 items 3 data follows
1.26 0.15 2.99
attribute "dep" string "positions"
#
# % Ppt Rep.by F Chem.
object 7 class array type float rank 0 items 3 data follows
0 0 0
attribute "dep" string "positions"
#
# 1985 field
object 8 class field
component "positions" value 1
component "id" value 2
component "CA" value 3
component "MG" value 4
component "K" value 5
component "SO4" value 6
component "%Ppt" value 7
attribute "series position" number 1985
```

The current visual program will still produce an image from the updated data file. However, the data that is displayed is always just the last component added, i.e., '%Ppt'. Remember that the server must be reset in order for the file to be re-imported.

Extend the program to allow the end-user to select which component (data value) is to be used for the visualization. In order to change which object is the 'data' component within the visual program, you must use the **Mark** module.

- Wire a **Mark** module between the **Import** and **AutoGlyph** modules. Place an **Inquire**, **Select**, and **Selector** on the canvas to the right of the **Import** and **Mark** modules. Wire them together as shown in Figure 15.9.

Figure 15.9
Component Selector VPE



- Set the *inquiry* input of **Inquire** to "component names".

The output from **Inquire** is a list of all the component names for the field. This list includes component 1, 'positions', and a new component constructed during the **Import**, 'box'. The **Select** module is used to select a subset of the list. For this example, you want to chop the first name and last name off the list. The list names are indexed from 0 to 7, so select 1 through 6 to pass the desired set of data component names to the **Selector**.

- Set the *which* input of **Select** to { 1 .. 6 }. Make sure to place spaces between the 1, the ellipsis, and the 6.

The list that is fed into the **Selector** now contains only the data component names. Note that the **Selector**'s menu list is not updated until you connect all the modules and execute the program once. The **Mark** module requires a string input to identify which component to copy into the 'data' component. Therefore, connect the right output of the **Selector** to the **Mark** module. The **Selector**'s left output is a value that represents the component's position in the list.

This same technique for selecting components can be very useful for data imported with the **ImportSpreadsheet** module. **ImportSpreadsheet** imports data as a table with each column as a

separate component of a field. The field contains no 'data' component for other modules to manipulate. The **Mark** or **Rename** module must be used to identify which component represents the 'data' of interest.

Now, add a description of 1986 data to the native DX file. The positions and id objects do not change, and the DX file format allows those objects to be used again for the second time slice without having to be recreated. Thus you only need add data objects that are different.

Create the new objects (components) at the end of the file for the next time slice.

Figure 15.10 *Second time slice*

```
#
# Begin the next time slice.
#
# CA values
object 9 class array type float rank 0 items 3 data follows
0.53 0.12 1.73
attribute "dep" string "positions"
# Mg values
object 10 class array type float rank 0 items 3 data follows
0.083 0.024 0.172
attribute "dep" string "positions"
# K values
object 11 class array type float rank 0 items 3 data follows
0.025 0.053 0.145
attribute "dep" string "positions"
# SO4 values
object 12 class array type float rank 0 items 3 data follows
1.10 0.42 2.38
attribute "dep" string "positions"
# % Ppt Rep.by F Chem.
object 13 class array type float rank 0 items 3 data follows
0.0 0.0 0.0
attribute "dep" string "positions"
#
# 1986 field
object 14 class field
component "positions" value 1
component "id" value 2
component "CA" value 9
component "MG" value 10
component "K" value 11
component "SO4" value 12
component "%Ppt" value 13
attribute "series position" number 1986
```

Notice that the component objects for 'positions' and 'id' are the same as the previous field. Reusing these objects instead of creating new (duplicate) ones saves on the memory used to define the OpenDX objects.

Create a final object that contains the two years as a series group, as shown in Figure 15.11.

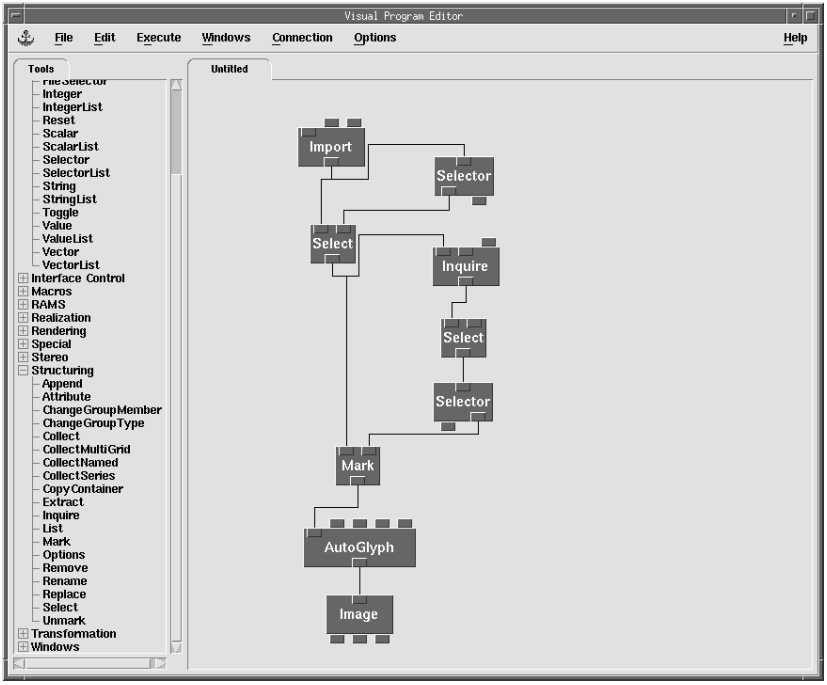
Figure 15.11 *Series group object*

```
#
# Define a series group containing the two members.
#
object "default" class series
member 0 position 1985 value 8
member 1 position 1986 value 14
#
end
```

If you try to run your visual program with this data set, you will now get an error because the **Inquire** requires a field, but now the data is defined as a group. You need to add another **Select** to choose which field (i.e., year) to display.

- Place a **Selector** and a **Select** between the two modules, wired to the **Import** as shown in Figure 15.12.

Figure 15.12
VPE with series selector



- Add two more years worth of data and expand the series group accordingly, as shown in Figure 15.13.

Figure 15.13 *Two more years of data*

```
#
# Begin the next time slice.
#
# CA values
object 15 class array type float rank 0 items 3 data follows
0.74 0.18 1.03
attribute "dep" string "positions"
# Mg values
object 16 class array type float rank 0 items 3 data follows
0.101 0.037 0.158
attribute "dep" string "positions"
# K values
object 17 class array type float rank 0 items 3 data follows
0.036 0.056 0.075
attribute "dep" string "positions"
# SO4 values
object 18 class array type float rank 0 items 3 data follows
1.37 0.91 2.20
attribute "dep" string "positions"
# % Ppt Rep.by F Chem.
object 19 class array type float rank 0 items 3 data follows
22.60 75.50 22.40
attribute "dep" string "positions"
#
# 1987 field
object 20 class field
component "positions" value 1
component "id" value 2
component "CA" value 15
component "MG" value 16
component "K" value 17
component "SO4" value 18
component "%Ppt" value 19
attribute "series position" number 1987
#
# Begin the next time slice.
#
# CA values
object 21 class array type float rank 0 items 3 data follows
0.80 0.17 1.40
attribute "dep" string "positions"
# Mg values
object 22 class array type float rank 0 items 3 data follows
0.119 0.026 0.159
attribute "dep" string "positions"
# K values
object 23 class array type float rank 0 items 3 data follows
0.092 0.047 0.055
attribute "dep" string "positions"
# SO4 values
object 24 class array type float rank 0 items 3 data follows
1.08 0.74 2.61
attribute "dep" string "positions"
# % Ppt Rep.by F Chem.
object 25 class array type float rank 0 items 3 data follows
60.10 42.20 14.60
attribute "dep" string "positions"
```

```
#
# 1988 field
object 26 class field
component "positions" value 1
component "id" value 2
component "CA" value 21
component "MG" value 22
component "K" value 23
component "SO4" value 24
component "%Ppt" value 25
attribute "series position" number 1988
#
#
# Define a series group containing the two members.
#
object "default" class series
member 0 position 1985 value 8
member 1 position 1986 value 14
member 2 position 1987 value 20
member 3 position 1988 value 26
#
end
```

The data is now defined as a series group containing a field object for each year. Since each field contains a positions component, you can use the **Connect** module to create a connections component, allowing OpenDX to interpolate between the points to smooth the depiction of the data.

Replace the **AutoGlyph** module with a **Connect** and **AutoColor** module and wire the network, as shown in Figure 15.14.

Although this manual process works, it is very tedious to construct. For example, imagine how much time that it would take to construct a native DX file for all of the collecting stations within the U.S. The key in constructing native file descriptions for large, common data formats is to complete enough of the manual process to develop an understanding of what the desired file must look like. Then, given a little time and programming expertise, a developer should be able to write a program that takes original input files as inputs, and creates a single, appropriate DX native file as output.

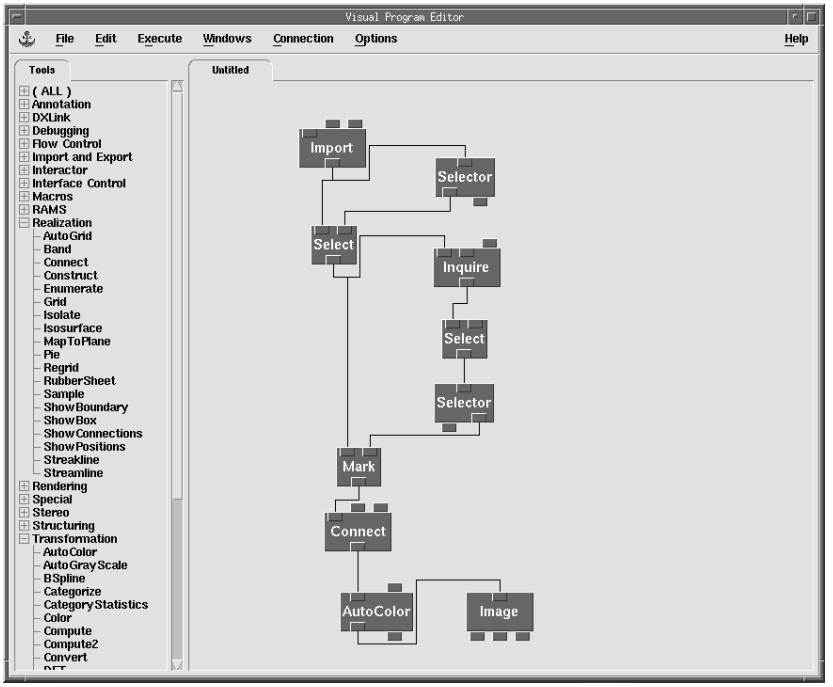


Figure 15.14
VPE with Connect

Conclusion

Understanding the hierarchical data structure of OpenDX provides the programmer with a better overall understanding of how OpenDX works. The native file format is a basic extension of the data structure and adds more flexibility to the software.

A native OpenDX file contains 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 either ASCII text or binary data that is referred to by the header section. The header section can reference other objects or data either in the same file or in other files.

The native OpenDX file format allows the user to transform output from computer simulations, Web pages, or other on-line sources into a format that is directly importable into OpenDX. This transformation can be done by hand, or implemented in a separate "filtering" program. A programmer can also tightly couple a filtering program by having it direct its formatted output to stdout instead of a file, then starting and running the program within OpenDX. To do this, use the **Import** module and enter "!program_name" as the Name input parameter.

Conclusion 16

The preceding chapters attempt to provide the reader the background knowledge and hands-on experience necessary to begin using OpenDX to develop non-trivial visualizations. However, what is illustrated in these chapters and their examples is just a fraction of the full power of OpenDX. The examples and exercises presented provide enough background for a typical user to independently use the software for visualization, but should be only a starting point for the serious user.

In covering various basic topics: basic terminology of data, the OpenDX user interfaces, OpenDX's extensive data importing facilities, and the OpenDX data model, this book provides the fundamental background needed to use OpenDX. In passing, the book also tries to help the user understand how the DX Executive manages the flow of DX data objects and how various modules transform objects to create imagery. The book also provides some hints on OpenDX programming, particularly in pointing out debugging modules and techniques that can help the OpenDX user understand how a data object changes after each module manipulates it.

What's next for the user? For additional understanding of OpenDX and the modules that are not presented in this text, examine the plethora of available sample programs included with the OpenDX distribution. The examples are by default installed in the "samples/programs" directory within the "dx" root directory. The Quickstart Guide, User's Reference, and User's Guide can also be of great assistance. Within the Quickstart Guide are the standard tutorial, information about the general array import format, and further help with the data prompter. The User's Guide provides an in-depth look at the data model, execution model, scripting language, and provides information about file formats and command line options. The User's Reference provides for quick lookup of each standard module's description. All this documentation is available in HTML format within the "dx" installation directory, or in other common formats at the opendx.org Web site. Also, much of the OpenDX documentation is available through the Help menu, so the user need not turn to external sources to get most questions answered.

OpenDX offers much more than presented in this book. The software is extremely extensible. It provides a solid standard foundation for visualization, along with a full set of application programming interfaces (APIs) that promote customization and extensions. The APIs help the user build his/her own custom transformations, packaged as modules, to develop custom user interfaces,

and to link to the Executive (DXLink) or just use the module transformations using DXCallModule. For more information on using OpenDX's APIs, refer to the Programmer's Reference manual as well as the programming examples available in the "samples" directory.

Since May 1999 when IBM announced its plans to open source Data Explorer, the user community has been steadily growing from the original commercial customer base to a much larger community of users and contributors. As with any large open source software system, on-going development of OpenDX relies on the contributions of the user community. Users can find more information about OpenDX on the Internet at www.opendx.org and www.research.ibm.com/dx. From there, interested users can join various e-mail lists and find additional community information.

Index

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W